5
Aspectos de implementação do micro-núcleo

Este capítulo tem como objetivo estabelecer algumas das principais decisões de projeto na implementação da nova versão da ferramenta Composer, mais especificamente do micro-núcleo. As tecnologias empregadas nesse desenvolvimento são discutidas e suas escolhas dentro do projeto são justificadas de acordo com os requisitos levantados no Capítulo 3. O micro-núcleo foi desenvolvido tendo como sustentação a arquitetura proposta no Capítulo 4, servindo como uma prova de conceito e validação dessa proposta.

Esse capítulo expõe os pontos de extensão que devem ser utilizados pelos plug-ins para acrescentar novas funcionalidades ao micro-núcleo. Discorre sobre cada uma de suas funções presentes na API, explicando sua forma de uso e as vantagens dessa maneira de extensão.

5.1.Aspectos do Projeto

O Composer I, detalhado no Capítulo 2, foi desenvolvido utilizando Java, trazendo naturalmente a vantagem da multiplataforma, mas que por ser uma linguagem interpretada, tem desvantagens em relação a desempenho. Para o desenvolvimento de plug-ins que lidem com a exibição de múltiplos tipos de mídia, ou até mesmo um pré-visualizador integrado, o desempenho é uma variável importante.

O Composer I faz uso de um exibidor de documentos NCL desenvolvido em Java, chamado de *NCL Emulator*. Esse exibidor utiliza como base para exibição de mídias o arcabouço *Java Media Framework* (JMF) (Oracle Inc., 2010). Devido a uma série de limitações desse framework, o *NCL Emulator* teve desenvolvimento descontinuado. As principais limitações que levaram esse decisão foram: a falta de suporte para vídeos codificados em MPEG-2 (ISO/IEC, 2000) e MPEG-4 (Eleftheriadis, 2001); falta de suporte para áudios codificados em MPEG-4 AAC (Eleftheriadis, 2001). Os problemas relacionados à codificação de mídias podem ser tratados com incorporação de plug-ins, mas diferente do

JMF os plug-ins não estão disponíveis em todas as plataformas. Além disso, existem críticas sobre a evolução e manutenção da JMF por parte da SUN, agravadas pela compra da SUN feita pela Oracle. Com isso, várias implementações alternativas começaram a surgir, sendo a principal deles *Freedom For Media in Java* (FMJ) (SourceForge, 2010).

Outras limitações conhecidas da JMF são: a impossibilidade de sobreposição de mídias estáticas. Por exemplo, imagens e textos não podem ser sobrepostos sobre mídias de vídeos; o controle de volume diferenciado para cada mídia de áudio e/ou vídeo; a falta de recursos, na atual API da JMF, para edição de mídias contínuas e a ausência para anotação de metadados. A Sun lançou pacotes da JMF, específicos a cada plataforma, com o objetivo de resolver problemas de performance, mas até o momento nem todas plataformas foram atendidas.

A fim de dar maior robustez e melhora no quesito de desempenho. O trabalho proposto será totalmente desenvolvido em C++, utilizando arcabouços que sejam multiplataforma. Este capítulo visa analisar as alternativas existentes (em C++) que não possuam as limitações da JMF, dando um embasamento teórico sobre as tecnologias e metodologias que serão utilizadas no trabalho proposto.

5.1.1.QT, GTK e GTKmm

Existe uma variedade enorme de arcabouços para o desenvolvimento de interfaces e manipulação de mídias. Os principais e mais utilizados são GTK (GNU, 2007), tendo o GTKmm sua versão C++, e o QT. Ambos arcabouços foram amplamente estudados e suas vantagens e desvantagens em relação um ao outro foram contrapostas. O arcabouço escolhido para o desenvolvimento desse projeto foi o QT. Algumas das vantagens que levaram a essa decisão serão expostas adiante.

O QT, diferente do GTK+, não está restrito ao desenvolvimento de interfaces gráficas. O QT é tido como um arcabouço completo para desenvolvimento de aplicações, fornecendo integradamente diversos recursos utilizados em qualquer tipo de aplicação. Essa diversidade de recursos disponível dá um poder de desenvolvimento ao programador que pode facilmente conectar

eventos de interface com banco de dados, conexão HTTP, entre outros. Na comunidade de desenvolvimento esse arcabouço é chamado de Java para C++.

A principal vantagem do QT é a existência de uma empresa de grande porte vinculada ao seu desenvolvimento, a Nokia. O histórico de lançamentos de novas versões e *binds* para novas linguagens de programação é muito mais rápido. Enquanto em um ano são lançadas 3 novas versões para o QT, o GTK encontra-se estagnado na versão 2.3 há mais de um ano. O QT é utilizado no sistemas de janelas KDE, enquanto o GTK é a base do GNOME. Em termos de aplicações ambos são parecidos. A documentação oficial do QT é extensa e detalhada, coberta de exemplos e tutoriais para cada módulo do seu arcabouço. Isso se deve ao fato da Nokia estar por trás do seu desenvolvimento e o a equipe de manutenção do QT ser cinco vezes a do GTK.

O QT tem suporte nativo para todas plataformas, inclusive para dispositivos móveis e/ou portáteis que utilizam Symbian, Windows CE/Mobile ou Maemo, e mais recentemente foi adicionado um *port* de QT para plataforma móvel do Google – Android. Além disso, a Nokia anunciou que o seu sistema portátil de tablets, denominado WebOS, é totalmente baseado no QT.

Além disso, todo seu ambiente de desenvolvimento é integrado, indo desde a criação da interface via *drag & drop,* conexão dos elementos da interface com funções no código, até a compilação e depuração da aplicação. Esse ambiente integrado de autoria e uma API mais completa dão mais produtividade ao programador. Em média uma aplicação escrita em QT tem 30% menos código do que se fosse escrita em GTK+.

O QT é licenciado através de uma licença dupla: para aplicações gratuitas e comerciais de código-aberto é utilizada a mesma licença GNU LGPL 2.1; para aplicações proprietárias fechadas é necessário adquirir uma licença comercial.

OS	Qt	GTK+
Windows XP	Nativo	Nativo
Windows Vista	Nativo	Nativo
Windows Mobile (CE)	Nativo	Não disponível
Mac OSX	Nativo	Parcialmente disponível
Linux/Unix	Nativo	Nativo
Symbian (S60)	Nativo	Não disponível
Android	Parcialmente disponível	Não disponível
Tablets	Nativo (WebOS)	Não disponível

Tabela 3 - Plataformas suportadas elo GTK+ e QT

5.1.2.Mecanismo de comunicação entre objetos

No mecanismo de comunicação apresentando no Capítulo 4, existe uma grande troca de mensagens entre os plug-ins e o micro-núcleo, dependendo da quantidade de plug-ins e o número de alterações que os plug-ins irão realizar no documento. Esse mecanismo é um gargalo potencial da arquitetura proposta, por conta disso existe alguns requisitos para a maneira como os objetos se comuniquem.

O micro-núcleo deve trabalhar de maneira igual e independente do número de plug-ins ligados a ele, pois esses plug-ins podem ser carregados e descarregados dinamicamente em tempo de execução. Ao enviar uma notificação de alteração no modelo para os plug-ins o micro-núcleo não deve se preocupar em manter uma lista desses plug-ins, o micro-núcleo só tem que gerenciar os grupos de entidades que estão presentes no modelo e enviar a notificação para os plug-ins interessados na entidade que foi alterada.

Pode-se comparar os grupos de entidades com grupos *multicast* onde o protocolo é responsável por entrada e saída de nós (plug-ins) do grupo e a fonte da mensagem só envia a notificação para um determinado grupo *multicast* e o protocolo é responsável por entregar essa mensagem somente para os nós pertencentes a esse grupo específico.

O micro-núcleo, ao enviar a notificação, não deve ficar esperando a resposta de cada plug-in que aquela alteração foi processada. Essa comunicação deve ser

assíncrona, de maneira que o micro-núcleo envie a notificação para os plug-ins e possa estar livre para lidar com novas modificações.

Dada essas restrições, o QT fornece um mecanismo de comunicação entre objetos chamado de *Signal/Slot*, uma das fortes razões da escolha do QT para esse projeto. No tradicional mecanismo de *callbacks*, o objeto que irá emitir a mensagem deve explicitamente fazer a chamada da função correspondente em cada um dos objetos que devem receber a mensagem. Para isso o emissor deve manter uma lista dos objetos, nesse caso os plug-ins. Além disso, para realizar a chamada no próximo objeto na lista é necessário esperar que o primeiro objeto retorne o resultado da função. Já no mecanismo de *Signal/Slot*, sinais são conectados a *Slots*, e o QT é responsável por manter a lista de conexões entre os objetos. Quando um sinal é disparado todos os *Slots* a ele associados são executados.

O mecanismo de comunicação via signal/slot é ilustrado na Figura 13. Observa-se que um sinal pode ser conectado a diferentes *Slots* que não precisam estar no mesmo objeto. Esse mecanismo é fracamente acoplado, visto que o objeto emissor não precisa saber o número de funções que serão chamadas, nem mesmo quais objetos irão tratar essa mensagem.

Existem dois tipos de conexão entre sinais e *Slots*: conexão bloqueante e conexão assíncrona. A conexão bloqueante deve ser utilizada quando o objeto emissor do sinal só prossegue sua execução após o *Slot* do objeto receptor tiver sido finalizado. A conexão assíncrona é exatamente a que será utilizada nesse projeto, o micro-núcleo (objeto emissor) do sinal irá prosseguir sua execução, independente de quando o objeto receptor (plug-ins) terminar de tratar o sinal. A vantagem da conexão assíncrona é que se previne que plug-ins que demandem mais tempo no tratamento se tornem um gargalo no sistema de comunicação.

Para fazer a conexão, a assinatura do sinal e do *Slot* devem ser iguais, garantindo a segurança de tipagem. Os *Slots* são funções comuns e podem ser sobrecarregadas e ser chamadas sem a necessidade do disparo de um sinal. É possível conectar quantos sinais forem necessários a um único *Slot*; mais do que isso, é possível ter uma cadeia de sinais - conectando um sinal diretamente a outro sinal.

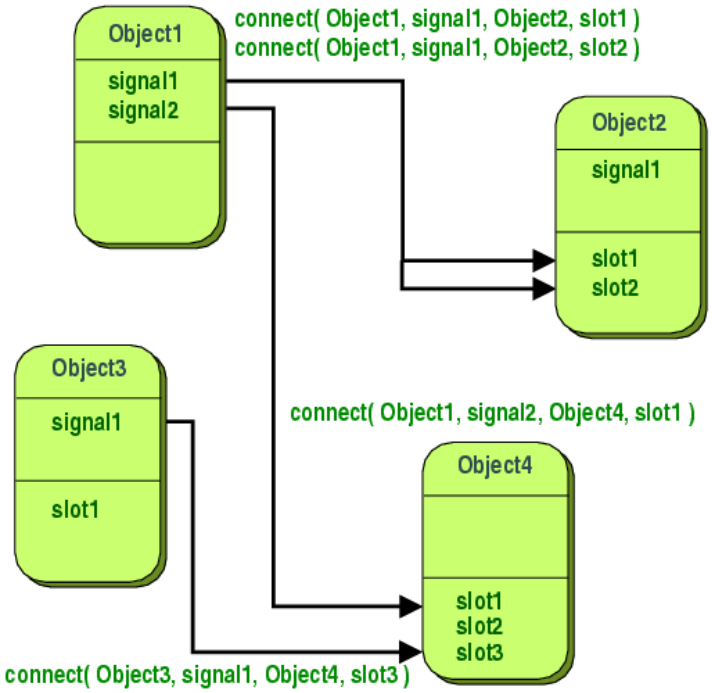

Figura 13 - Mecanismo de comunicação signal/slot

5.2. Aspectos de implementação

Os aspectos de projeto e implementação apresentados nesse capítulo dizem respeito ao desenvolvimento do micro-núcleo e os seus pontos de extensão. O Capítulo 6 traz detalhes e informações do acoplamento do Composer II com esse micro-núcleo e os plug-ins, mostrando que o desenvolvimento do micro-núcleo é totalmente independente da interface gráfica da ferramenta.

5.2.1.API de extensão do micro-núcleo

Os plug-ins incorporados a esta arquitetura devem interagir com o micro-núcleo através de uma interface de comunicação ortogonal. Essa interface deve ser simples e direta, habilitando um desenvolvedor terceiro a implementar uma nova funcionalidade de maneira rápida. Além disso, essa interface deve abstrair a maneira como o plug-in é carregado e se comunica com o micro-núcleo. O desenvolvedor só precisa focar o desenvolvimento do plug-in seguindo a interface, o restante deve ser responsabilidade do micro-núcleo.

O arcabouço QT já dispõe de um mecanismo próprio de extensão de aplicações utilizando plug-ins. Esse mecanismo é chamado de *QtPlugins*, e será

utilizado no desenvolvimento do micro-núcleo. Utilizando *QtPlugins*, o desenvolvedor de um novo plug-in implementa uma aplicação QT normalmente, fazendo uso de suas ferramentas de desenvolvimento e design de interface gráfica. A fim de transformar essa aplicação QT em um plug-in, o desenvolvedor precisa especificar quais classes da sua aplicação implementam as interfaces disponibilizadas pelo micro-núcleo. O micro-núcleo será o responsável por identificar essas classes dentro da implementação do plug-in, carregar o plug-in sobre o micro-núcleo e fazer as vias de comunicação entre os plug-ins e o micro-núcleo.

Uma limitação das ferramentas atuais é a impossibilidade da edição simultânea em mais de um documento. Isso se deve ao fato dessas ferramentas só lidarem com um modelo de documento interno de cada vez. Essa limitação foi resolvida por essa arquitetura, através da criação de duas interfaces: *IPluginFactory* e *IPlugin*. A *IPluginFactory* é uma interface que define características básicas dos plug-ins, como nome, ícone, identificador, etc. O mais importante é que essa interface define como criar e destruir instâncias de plug-ins, permitindo o micro-núcleo lançar várias instâncias do mesmo plug-in. A diferença é que cada instância está diretamente ligada a uma instância de documento.

A **Listagem 1** apresenta a interface *IPluginFactory*. Essa interface possui funções para identificar (**3**), nomear (**4**) o plug-in dentro do micro-núcleo, assim como uma função para requisição de ícone (**5**) para ser utilizado pela *ToolGUI*, na interface gráfica. A função *createPluginInstance*() retorna uma instância da classe que implementa a interface *IPlugin*, de posse dessa instância o micro-núcleo pode realizar as conexões para troca de mensagens com o plug-in. Enquanto a função *releasePluginInstance*() deve ser chamada para liberar uma instância do plug-in passada como parâmetro, desocupando a memória quando o documento não está mais sendo editado.

```
1: IPlugin* createPluginInstance();
2: void releasePluginInstance (IPlugin *);
3: QString getPluginID();
4: QString getPluginName();
5: QIcon getPluginIcon();
6: QWidget* getPreferencePageWidget();
7: void setDefaultValues();
8: void applyValues();
```

Listagem 1 - Interface do IPluginFactory

A **Listagem 2** apresenta a interface *IPlugin*. Essa interface define as funções relacionadas com a troca de mensagens entre o plug-in e o micro-núcleo. As funções iniciadas com *onEntity* são slots que serão chamados pelo micro-núcleo a fim de notificar o plug-in de uma alteração no documento ao qual essa instância está atrelada. Por exemplo, *onEntityAdded(pluginID, Entity*)* notifica que uma entidade (*Entity**) foi adicionada no modelo do documento através de uma requisição do plug-in identificado por *pluginID*. Vale lembrar que a referência *Entity** é a mesma existente no modelo. Dessa maneira, o plug-in pode ter sua estrutura de dados própria para guardar as entidades do modelo.

A única função que difere das outras é a de remoção de uma entidade no modelo. Visto que as referências das entidades estão também de posse dos plug-ins, é necessário avisar primeiramente a esses plug-ins que essa entidade será removida (*onEntityAboutToRemove(Entity*)*), para somente depois fazer sua remoção no modelo.

As funções terminadas em *Entity* são sinais emitidos pelo plug-in para requisitar ao micro-núcleo uma modificação no modelo. Por exemplo, *addEntity(type, parentId, atts, force)* solicita ao micro-núcleo que uma entidade do tipo dado como *type* seja adicionada como filha da entidade identificada como *parentId*, com o mapa de atributos e valores *atts*. O último parâmetro *force* diz respeito à consistência do modelo. No processo de edição, muitas vezes o modelo se encontra semanticamente incorreto. Dada uma alteração, o plug-in pode querer forçar que essa alteração seja feita mesmo acarretando um modelo inválido (*force=true*), ou que essa alteração só seja feita se o resultado for um modelo válido (*force=false*).

```
1:  QWidget* getWidget();
2:  bool save();
3:  void updateFromModel();
4:  void onEntityAdded(pluginID, Entity *);
5:  void onEntityAddError(QString error);
6:  void onEntityChanged(pluginID, Entity *);
7:  void onEntityChangeError(error);
8:  void onEntityAboutToRemove(Entity *);
9:  void onEntityRemoved(pluginID, entityID);
10: void onEntityRemoveError(error);
11: void addEntity(type, parentEntityId, atts, force);
12: void editEntity(Entity *, atts, force);
13: void removeEntity( Entity *,force);
```

Listagem 2 - Interface IPlugin

Além das funções relacionadas com troca de mensagens, essa interface possui uma função *save()*, a qual é chamada pelo micro-núcleo antes de liberar o plug-in da memória. Com isso, o plug-in pode salvar características próprias relacionadas aquele documento, como cor e tamanho de fonte. Um plug-in pode ser carregado dinamicamente e no momento que se dá a edição de um documento. Para isso, o micro-núcleo chama a função *updateFromModel()* para notificar o plug-in que o documento a ele atrelado já estava em edição e que é preciso atualizar sua interface de acordo com o modelo de documento.

Uma aplicação feita normalmente em QT é um conjunto de objetos de interface chamados de *QWidget*, inclusive a sua janela principal. Sendo assim, a função *getWidget()* deve retornar ao micro-núcleo qual desses objetos de interface do plug-in devem ser incorporados à interface gráfica da ferramenta.

5.2.2.Diagramas de classe

Para evitar que o plug-in tenha que enviar uma mensagem para o micro-núcleo para que esse devolva uma propriedade de uma entidade, foram definidas duas operações: de manipulação e de consulta. As operações de consulta podem ser feitas diretamente pelos plug-ins sobre o objeto que representa a entidade ou o

documento. Como discutido, cada plug-in terá uma referência para o documento e para as entidades que vão sendo criadas ao longo da edição.

A Figura 14 apresenta o diagrama de classe do modelo interno do micro-núcleo, como pode ser visto este foi otimizado e enxuto ao máximo, por questões de manutenção e performance. Um documento é composto basicamente de entidades, podendo estas serem ou não aninhadas, uma localização dentro do sistema de arquivos e uma referência para o projeto em que está contido.

As operações de consulta são as marcadas por *get* no inicio. Por exemplo, a partir da sua referência para o documento, um plug-in pode requisitar a localização desse documento, em qual projeto o mesmo está inserido e até mesmo buscar uma entidade dado um determinado identificador. Enquanto as operações marcadas como *set* em seu inicio são utilizadas pelo micro-núcleo para fazer a manipulação do documento.

A Entidade dentro do modelo é composta por um mapeamento de atributos, onde a chave é o nome do atributo, uma referência direta para o seu pai, permitindo rápido acesso transversal no modelo caso o plug-in precise. Além disso, cada entidade deve possuir um identificador único dentro do modelo e um tipo específico. O identificador serve como base para uma rápida localização da entidade dentro do modelo, otimizando funções de modificação e deleção, enquanto o tipo está relacionado com o filtro ao qual o plug-in poderá solicitar.

A Entidade fornece para os plug-ins uma API para consulta. Dentro dessa API existem funções para requisitar um determinado atributo, saber se um atributo específico existe nessa entidade, e inclusive pegar os iteradores de atributos e filhos para serem utilizados da maneira que o plug-in julgar conveniente. É importante ressaltar que somente o micro-núcleo tem a permissão de modificar, criar ou remover uma entidade do modelo, por isso as funções: *deleteChild*, *removeChild*, *setAttribute*, *setType*, *setUniqueId*, entre outras funções de manipulação não estão visíveis para os plug-ins.

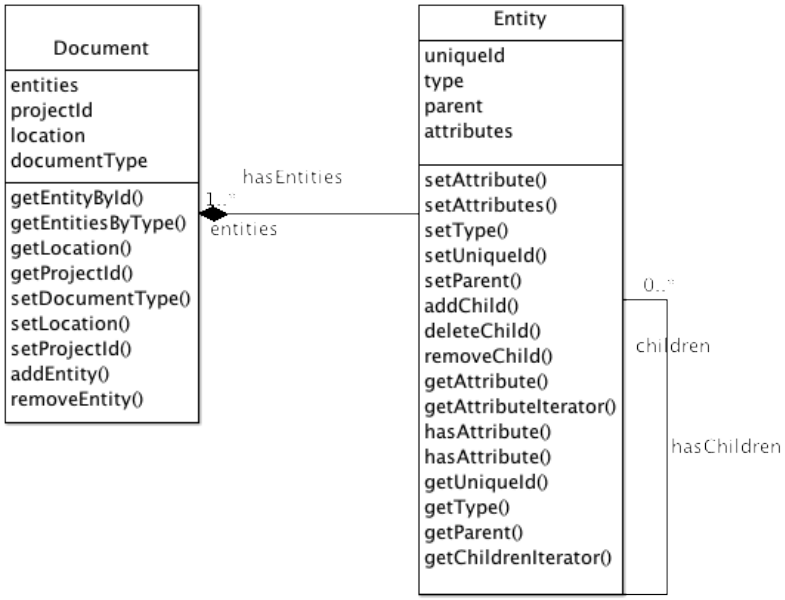

Figura 14 - Diagrama de Classe do modelo

A Figura 15 abaixo apresenta o diagrama de classe do micro-núcleo. Cada uma dessas classes está representando um módulo dentro da arquitetura apresentada no Capítulo 4. Os módulos omitidos (*CoreManager* e *ModelFacade*) não dizem respeito ao micro-núcleo em si, mas como esse micro-núcleo conversa tanto com a *ToolGUI* quanto o *CoreModel*. Tais módulos são apresentados no Capítulo 6, onde é discutida a maneira que o Composer II foi acoplado ao micro-núcleo.

A classe *PluginControl* é responsável pelas operações relacionadas aos plug-ins: carregamento, descarregamento dinâmico, assim como a criação e deleção de novas instâncias de plug-ins. Essa classe é composta por uma lista de fábricas de plug-ins e suas respectivas instâncias. Para cada instância criada do plug-in é necessária a criação de um *TransactionControl* associada aquela instância. É através dessa associação que é possível a edição simultânea de vários documentos, um requisito inexistente nas ferramentas de autoria atuais.

A classe *TransactionControl* é a única classe dentro do micro-núcleo que realiza operações de manipulação sobre o modelo. Cada instância dessa classe tem um *Document* associado. Esse *Document* é a mesma referência repassada para os plug-ins em sua criação. Dessa forma, quando o *TransactionControl* realiza a modificação no *Document*, essas modificações estão imediatamente disponíveis

para os plug-ins, mas esses só saberão que tal modificação foi realizada ao receber a notificação de volta do *TransactionControl*.

As funções marcadas com *on* em seu início são os slots que serão acionados pelos plug-ins para requisição de modificações, enquanto as funções marcadas como *entity* em seu início são sinais que irão notificar os plug-ins que tal modificação já foi realizada.

A classe *DocumentControl* tem como objetivo fazer o controle dos documentos que estão sendo editados no momento. É essa classe que recebe solicitações de abertura, fechamento ou remoção do documento.

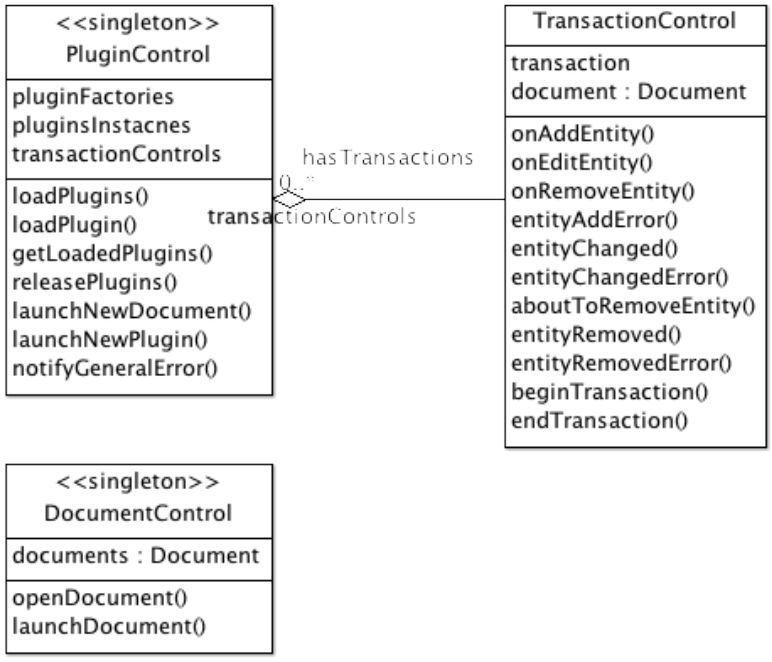

Figura 15 - Diagrama de classe do micro-núcleo

As classes *PluginControl* e *DocumentControl* seguem o padrão *singleton*, visto que somente uma instância dessas classes deve ser permitida. Os plug-ins carregados, assim como os documentos abertos devem ser únicos e mantidos de maneira centralizada, visando consistência dentro do micro-núcleo. Além disso, as respectivas instâncias dessas classes devem ser as mesmas nos diferentes lugares dentro do micro-núcleo em que essas são requisitadas.

5.2.3.Diagramas de sequência

O diagrama de classe apresenta apenas a composição das classes em termos de atributos e operações, mas não mostra como essas classes (suas instâncias) se

comunicam para realizar as tarefas requisitadas pelo autor. Os diagramas de sequência que serão apresentados têm como objetivo explicar a maneira como essas classes vão conversar e quais são as classes envolvidas nas tarefas de: carregamento inicial de plug-ins; criação de um novo documento (modelo); e requisição de alteração nesse modelo por um plug-in.

O diagrama de sequência da Figura 16 apresenta as classes envolvidas no carregamento inicial dos plug-ins. É importante ressaltar que esse carregamento é feito assim que se inicia a ferramenta, sem a interação do usuário (autor), mas também pode ser realizada pelo autor a qualquer momento.

A *ToolGUI* (interface da ferramenta) é requisitada (*initPlugins*) para fazer inicialização dos plug-ins. Como o *ToolGUI* não conversa diretamente com o micro-núcleo, é preciso fazer uma chamada (*loadPlugins*) à interface *CoreManager*. Essa interface então identifica qual dos módulos dentro do micro-núcleo irá ser responsável por tratar aquela requisição, nesse caso o *PluginControl*. O *CoreManager* repassa essa requisição (*loadPlugins(Path)*) para o *PluginControl*, mas com o caminho, dentro do sistema de arquivos, da localização dos plug-ins. Nesse momento, *PluginControl* varre o diretório passado como parâmetro e requisita (*loadPlugin(name)*) para o sistema de plug-ins do QT (*QtPlugins*) uma instância do plug-in. O *QtPlugins* cria essa instância, que é retornada para o *PluginControl*.

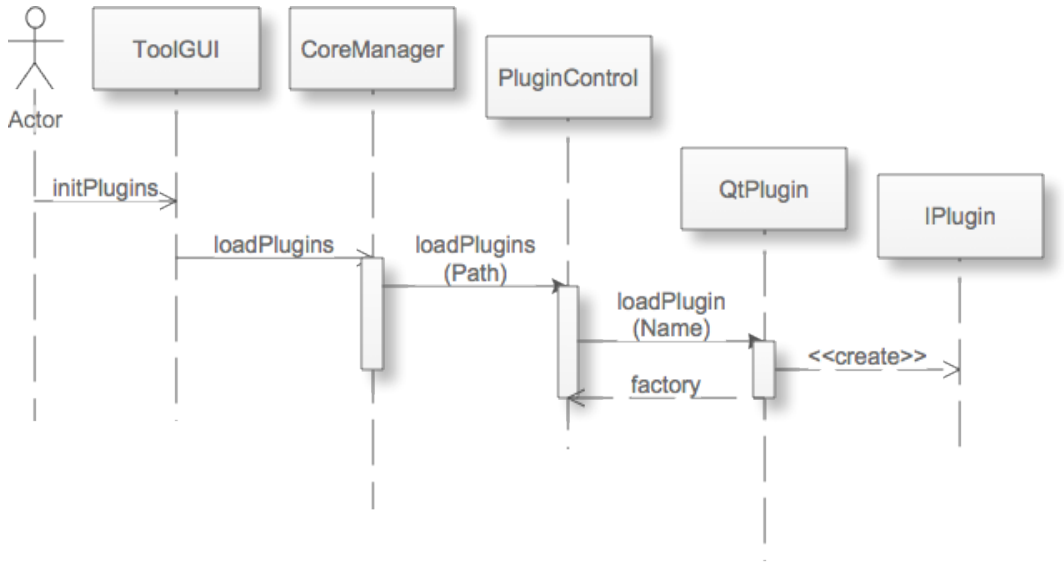

Figura 16 - Diagrama de sequência do carregamento inicial de plug-ins

A Figura 17 apresenta o diagrama de sequência, quando o autor solicita a abertura de um documento. As classes que dizem respeito à interface gráfica (*ToolGUI* e *CoreManager*) foram deixadas de fora por limitação de espaço na figura. A partir do ponto que o *DocumentControl* recebe a solicitação de abertura do documento, esse verifica se o documento já está aberto e, caso não exista um modelo para aquele documento, o *DocumentControl* solicita sua criação. De posse da instância do documento, uma chamada ao *PluginControl* é realizada (*launchNewDocument*) tendo como parâmetro o *Document* recém-criado.

Para cada fábrica de plug-in existente no *PluginControl* é feita uma solicitação de criação (*createPluginInstance*) da instância do plug-in correspondente. No caso da Figura 17, o *LayoutFactory* e *LayoutPlugin* são exemplificados. Após a criação das novas instâncias dos plug-ins é necessária a criação do *TransactionControl*, que também será atrelado ao modelo de documento recém-criado. O *PluginControl,* de posse de todas as instâncias, pode realizar a conexão das vias de comunicação entre e o *TransactionControl* e o *LayoutPlugin*.

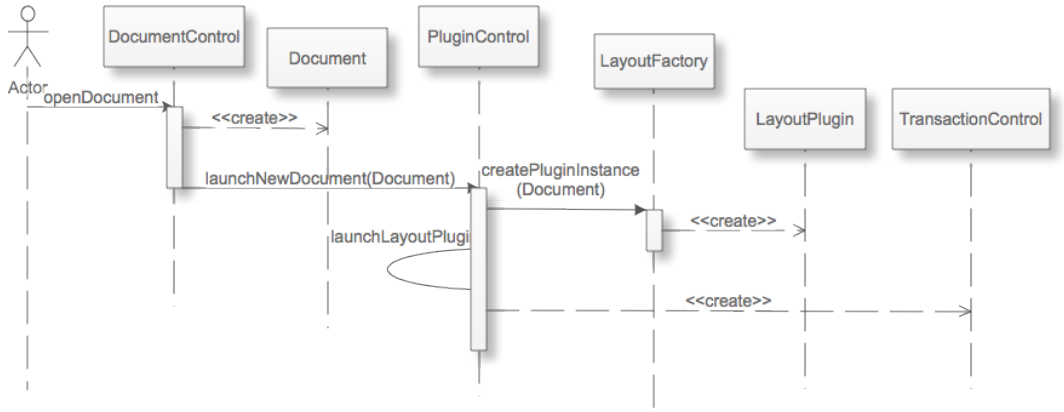

Figura 17 - Diagrama de sequência para abertura de documento

O diagrama de sequência apresentado na Figura 18 diz respeito à solicitação da criação de uma nova entidade no modelo, no caso uma nova região. A partir do momento que um novo documento é lançado e as instâncias dos plug-ins são dispostas na interface gráfica da ferramenta, o autor passa a interagir diretamente com o plug-in.

Nesse caso, o autor define visualmente uma nova região que o *LayoutPlugin* interpreta como sendo a solicitação para criação de uma nova região

(*createRegion*). O *LayoutPlugin* dispara o sinal *addEntity* para o micro-núcleo, notificando-o da requisição do autor. Esse sinal é recebido pelo *TransactionControl*. Por sua vez, faz a chamada de alteração no *Document* que cria a *Entity* e a adiciona no modelo.

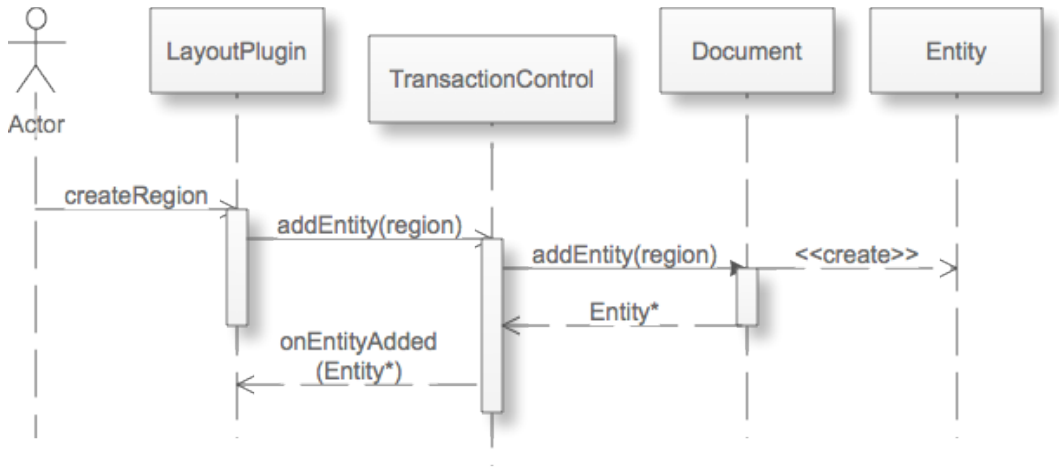

Figura 18 - Diagrama de sequência para criação de uma nova entidade

O *TransactionControl* compartilha a mesma instância do *Document* presente no *LayoutPlugin*, dessa maneira ao alterar o seu *Document* automaticamente estará alterando o *Document* do *LayoutPlugin*. O *TransactionControl* só precisa notificar a todos os plug-ins (inclusive o *LayoutPlugin*) que uma nova entidade foi adicionada, para isso ele dispara o sinal *onEntityAdded(Entity*)* que será recebido por todos os plug-ins conectados àquele modelo (*Document*).