3
Requisitos não-funcionais de ferramentas de autoria hipermídia

Na literatura são vários os trabalhos que discutem os requisitos funcionais desejáveis em uma ferramenta de autoria (Azevedo, Lima, Neto, & Texeira, 2009) (Bulterman & Hardman, 2005) (Bulterman D. C., Hardman, Jansen, Mullender, & Rutledg, 1998), (Guimarães, 2007), (Vazirgiannis, Kostalas, & Sellis, 1999) (Jourdan, Roisin, & Tardif, 2000). O trabalho em (Soares Neto & Soares, 2009) avaliou um grupo grande de autores hipermídia utilizando a linguagem NCL. Diferentes ferramentas de autoria foram apresentadas a esses autores (não restrito a ferramentas voltadas para NCL), dentre as quais uma versão anterior do *NCLEclipse* e o *Composer* em sua versão também anterior.

Não foi surpresa que a maioria dos programadores, especialistas com conhecimento em NCL, reportassem que as funcionalidades relacionadas à autoria textual, que eles consideram mais fáceis de utilizar do que as gráficas, deveriam ser ampliadas. Entretanto, esses mesmos programadores também consideram o uso de visões gráficas em menor extensão, em particular a visão de leiaute rudimentar presente no *NCLEclipse* para especificar a posição espacial inicial dos objetos de mídia.

Também não foi surpresa que praticamente todos os produtores de conteúdo iniciantes (comunicadores, artistas, designers e cidadãos comuns, sem qualquer especialidade voltada à informática ou produção de conteúdo) necessitam de uma ferramenta de autoria que abstraia toda sintática e semântica da linguagem de autoria, por meio do uso de visões gráficas. No entanto, à medida que o conhecimento desse grupo de autores sobre a linguagem aumentava, eles procuravam mapear as abstrações visuais, inseridas pela ferramenta, em elementos do código-fonte. Observou-se que a cada aplicação desenvolvida esses autores procuram mais e mais o uso da visão textual e, não encontrando as funcionalidades existentes em ferramentas puramente textuais, como o *NCLEclipse*, acabavam deixando o uso do *Composer*, migrando para o

NCLEclipse, embora com as críticas de falta de suporte visual para o desenvolvimento das aplicações.

3.1.Customabilidade e extensibilidade

A existência de diferentes perfis de autores, desde usuários sem qualquer conhecimento de programação até programadores especialistas, e a mudança nas necessidades desses autores à medida que adquirem conhecimento sobre a linguagem de autoria ou aumentam a complexidade de suas aplicações, trazem os primeiros requisitos não-funcionais para uma ferramenta de autoria: **customabilidade** e **extensibilidade**.

A ferramenta deve ser customizável no sentido de que o autor possa customizar a ferramenta de maneira a trabalhar com as visões que ele está mais confortável no seu estágio atual de expertise. Extensível no sentido que a ferramenta está preparada para receber novas funcionalidades e o autor possa ir agregando essas funcionalidades à medida que forem necessárias e moldá-las ao seu estágio de desenvolvimento corrente. Ao ser extensível, tal ferramenta abre espaço para a incorporação de novas técnicas de autoria que ainda estão para ser desenvolvidas.

Em uma ferramenta de autoria, cada visão tem o seu próprio modelo conceitual de dados de alto nível de abstração. Ao mesmo tempo, a ferramenta possui um modelo de dados central onde todas as ações de edição provenientes de cada visão são realizadas. Esse modelo usualmente é de mais baixo nível de abstração para ser mais apropriado a manipulações pela máquina de execução. Diferentes modelos de dados podem causar muitos problemas não-funcionais.

Usualmente, a distância semântica entre o modelo de dados central da ferramenta e o modelo de dados interno de uma visão dificulta o processo de tradução entre eles. Por conta desse problema, quase todas as ferramentas não são extensíveis, trabalhando com um número limitado de visões. Esse é o caso de todas as ferramentas apresentadas no Capítulo 2, à exceção do Composer. Mais ainda, muitas ferramentas executam a tradução em apenas uma via, sem permitir a sincronização entre os modelos internos das visões (a mudança em uma visão não é refletida automaticamente nas demais). Esse, por exemplo, é o caso das visões textuais nas ferramentas *LimSee2* e *GRiNS* apresentados no capítulo anterior. Em geral, as ferramentas consomem grandes recursos computacionais, requerendo

plataformas de ponta ou trabalhando com grandes atrasos, comprometendo seu uso nas edições em tempo real.

Como mencionado, praticamente todas as ferramentas apresentadas no Capítulo 2 trazem um número fixo de visões gráficas que mesmo combinadas e integradas não cobrem toda a expressividade da linguagem e acabam restringindo o autor. A única exceção é o *Composer*, que permite novas visões, criadas por terceiros, acopladas à ferramenta através de uma API pré-definida. Essa API é oferecida como uma classe abstrata em Java que pode ser herdada pelas novas visões. A abordagem, porém, traz muitas restrições ligadas a outros aspectos não-funcionais aos quais o *Composer* não dá suporte.

3.2.Portabilidade e escalabilidade

A Figura 10 apresenta o mecanismo de sincronização entre as visões no *Composer*, tendo por base um modelo central duplo (JavaNCM e DOM) e um modelo interno específico de cada visão. Para definir uma nova visão, é necessário definir um par de compiladores, que fazem o processo de tradução entre o modelo central JavaNCM e o modelo interno da visão. Quando uma visão específica modifica uma parte do documento NCL, o seu **observador** (obs1 na figura) é notificado. O observador então chama o seu compilador associado a fim de refletir as mudanças do modelo interno da visão no modelo central (JavaNCM) e, consequentemente, no modelo DOM. Os observadores (obs2 na figura) das demais visões são então notificados sobre a modificação no modelo central, quando então acionam seu compilador associado para traduzir do modelo central para seu modelo interno, atualizando, assim, todas as visões.

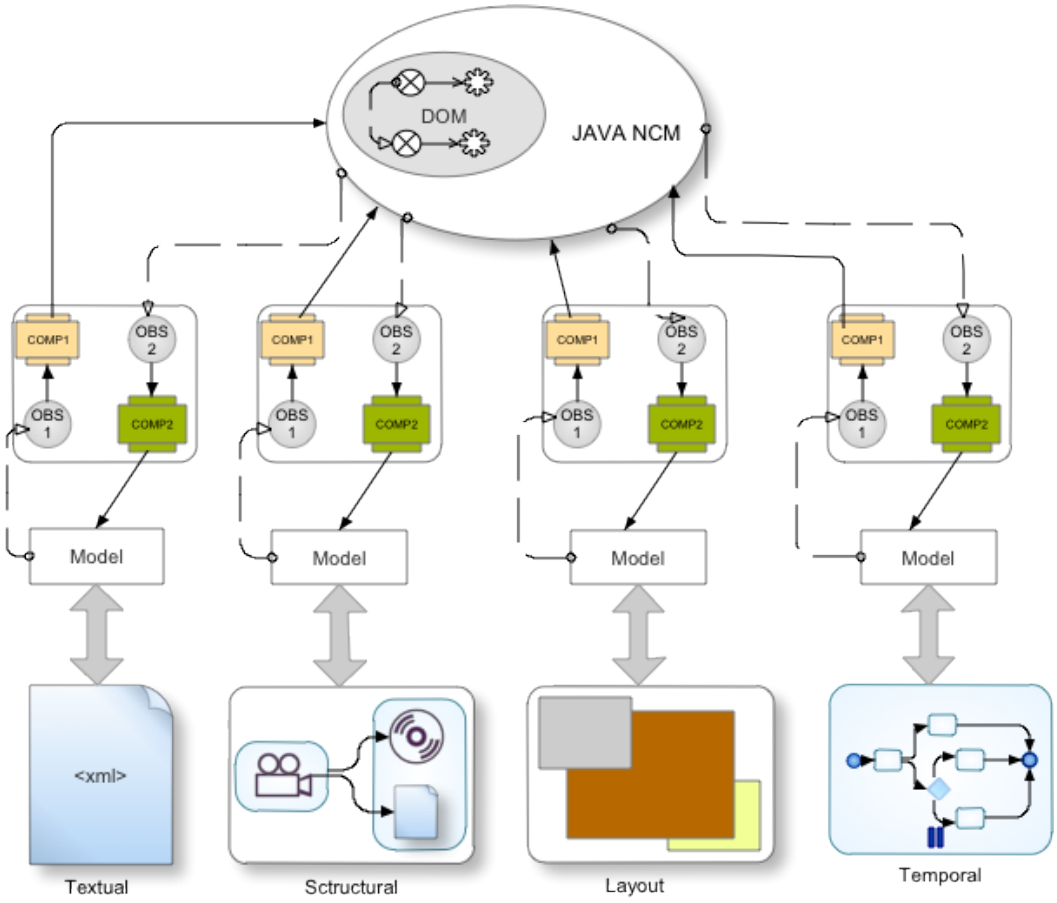

Figura 10 – Mecanismo de sincronização das visões no Composer I

É importante ressaltar que o mecanismo de sincronização adotado pelo Composer não é incremental. Uma pequena mudança acarreta o acionamento de N processos de compilação, onde N é o número de visões acopladas na ferramenta. O procedimento é pesado e não escalável, podendo rapidamente se tornar inviável, à medida que a aplicação sendo desenvolvida vai crescendo. Somente com as visões originais acopladas no *Composer,* o atraso causado pelas traduções acaba degradando a qualidade de experiência do autor, exceto em documentos NCL de pequeno porte. Dependendo do poder de processamento, os atrasos podem ser um gargalo até mesmo nesses documentos. Em outras palavras, a ferramenta também não é portável para plataformas de baixo desempenho.

Portabilidade é outro requisito não-funcional desejável, bem como **escalabilidade.** Uma vez que se quer uma ferramenta de autoria que permita não apenas aos produtores de conteúdo convencionais, mas também ao próprio telespectador desenvolver aplicações (para as redes de TV social), a possibilidade de ter a ferramenta customizada e possível de ser executada em plataformas

(receptores) de baixo custo e, portanto, com poucos recursos, é desejável. Assim, ser **portável** é um requisito a ser cumprido, e mais, não apenas para plataformas de baixo custo, mas com diferentes recursos de hardware e software (sistema operacional). A ferramenta deve também ser **escalável**, no sentido de permitir o desenvolvimento desde aplicações de pequeno porte, até aquelas de alta complexidade.

Ainda com relação aos requisitos de customabilidade, extensibilidade e portabilidade, uma ferramenta de autoria precisa ser **adaptável** não só aos diferentes autores, mas também ao ambiente de produção ao qual está integrada. A ferramenta deve ser customizável a fim de poder utilizada nos diferentes sistemas de armazenamento e de transmissão. Por exemplo, no caso da Web, a aplicação desenvolvida em geral é armazenada em um conjunto de arquivos em um servidor remoto; em um ambiente de TVD terrestre é transmitida através do fluxo de transporte – MPEG-2 TS; em um ambiente de serviços IPTV pode utilizar sistemas de transporte baseados no protocolo RTP (Internet Engineering Task Force (IETF), 2003) e fluxo FLUTE (Internet Engineering Task Force (IETF), 2004).

3.3.Desempenho

Desempenho permeia os diversos outros requisitos não-funcionais. Voltando à ferramenta Composer, diferentes motivos justificam o problema de seu desempenho mencionado nos parágrafos anteriores. Primeiro, a ferramenta foi implementada em Java. Essa decisão de implementação trouxe como principal vantagem a portabilidade para multiplataforma provida pelo Java, mas também trouxe a ineficiência no gerenciamento de memória e processamento de uma linguagem interpretada. Segundo, *Composer* usa dois modelos centrais de documento: a árvore DOM e o JavaNCM, este último é uma representação em Java do modelo NCM (*Nested Context Model*) (Soares & Rodrigues, 2005). Os dois modelos têm basicamente as mesmas informações. O uso da árvore DOM é justificado por ser um passo natural na tradução do documento físico NCL no modelo JavaNCM; e para manter a estrutura NCL, visto que a tradução JavaNCM para NCL não é um para um, ou seja, um modelo NCM pode gerar diferentes documentos NCL. Entretanto, manipular a árvore DOM e seus elementos é

custoso do ponto de vista computacional. O terceiro motivo é fundamental, no *Composer* alterações de autoria em uma visão não são refletidas nas demais visões de uma forma incremental, como já mencionado. Qualquer pequena alteração em uma visão causa a recompilação de todo o documento em todas as demais visões.

Note assim que, permeando todos os requisitos anteriormente mencionados está então o requisito não-funcional de **eficiência**. Eficiência é um requisito necessário tanto para minimizar o atraso no processo de sincronismo entre as visões, quanto no suporte à **escalabilidade** e **extensibilidade**. Com relação à minimização do retardo no processo de edição, o requisito de desempenho é essencial para se permitir a edição em tempo real, isto é, em tempo de exibição da aplicação. Além disso, eficiência é um fator essencial em plataforma com recursos escassos.

3.4.Confiabilidade e Adaptabilidade

A **adaptabilidade** da ferramenta deve levar em conta também o perfil do autor. Uma mesma visão da ferramenta pode ser incompleta ou proibitivamente complexa para um determinado perfil de autor ou aplicação. Por exemplo um comunicador parece se satisfazer com uma visão estrutural seguindo um paradigma de "StoryBoard", enquanto um designer prefere trabalhar em uma visão um pouco mais completa, arrastando e posicionando visualmente os objetos de mídia ao longo de uma linha do tempo.

Finalmente, nenhum dos requisitos funcionais e não-funcionais é importante em um sistema de autoria que não provenha um alto grau de confiabilidade. Tolerância a falhas é um requisito essencial, principalmente em um sistema extensível com visões desenvolvidas por terceiros. Falhas em uma visão devem ser confinadas a essa visão sem afetar o sistema como um todo. Mecanismos de recuperação e prevenção devem impedir que o autor tenha em risco seu trabalho em desenvolvimento.

3.5.Resumindo

A Tabela 2 resume e define os requisitos não-funcionais levantados neste capítulo.

Certamente os requisitos não são ortogonais e o atendimento a muitos deles é uma solução de compromisso. Assim, eles são listados na ordem de importância que foi considerada no desenvolvimento da prova de conceito, da arquitetura proposta no Capítulo 4, desenvolvida no Capítulo 5: a nova versão da ferramenta *Composer*.

Outros requisitos não-funcionais ainda existem, mas não foram considerados relevantes na realização do Capítulo 5. Entre eles está a **adaptabilidade** da ferramenta a diferentes linguagens de programação, ou seja, tendo como resultado uma especificação em diferentes linguagens. Outro requisito, no entanto, foi levado em consideração, a **manutenabibilade**, que inclui a facilidade de customização, de inclusão de novas visões ou novos sistemas de transmissão/armazenamento das aplicações.

Requisito não-funcional	**Descrição e objetivo**
Desempenho	Eficiência no uso de memória e CPU de forma a: proporcionar baixo retardo na sincronização de visões; viabilizar o uso em plataforma com poucos recursos; e possibilitar o uso eficiente de muitas visões e sistemas de transmissão/armazenamento.
Escalabilidade	Suporte ao desenvolvimento de aplicações, desde as mais simples até as mais complexas; possibilitar a adição de novas funcionalidades sem piorar o desempenho.
Customabilidade	Possibilidade de customização da ferramenta de forma que seu usuário possa trabalhar com as visões com que ele está mais confortável no seu estágio atual de expertise, e em diferentes sistemas de transmissão/armazenamento de aplicações.
Extensibilidade	Capacidade de se adicionar novas visões de autoria e novos sistemas de transmissão/armazenamento de aplicações.
Portabilidade	Possibilidade de execução em diferentes plataformas de hardware, desde plataformas com poucos recursos computacionais, e em diferentes plataformas de software (em

	particular, diferentes sistemas operacionais).
Confiabilidade	Tolerância a falhas, impedindo que falhas em uma visão, possivelmente implementada por terceiros, possa afetar (se propagar em) a ferramenta como um todo.
Manutenabilidade	Facilidade de de inclusão de novas visões ou novos sistemas de transmissão/armazenamento das aplicações, e de atualização da ferramenta.

Tabela 2 - Requisitos não-funcionais e sua descrição