UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

GEXT JS: FERRAMENTA VISUAL PARA FACILITAR O USO DO FRAMEWORK EXT JS

PAULO EDUARDO LEICHT

BLUMENAU
2016

PAULO EDUARDO LEICHT

GEXT JS: FERRAMENTA VISUAL PARA FACILITAR O USO DO FRAMEWORK EXT JS

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dr. Matheus Carvalho Viana - Orientador

BLUMENAU
2016

GEXT JS: FERRAMENTA VISUAL PARA FACILITAR O USO DO FRAMEWORK EXT JS

Por

PAULO EDUARDO LEICHT

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Matheus Carvalho Viana, Doutor – Orientador, FURB
Membro:	Prof. Everaldo Artur Grahl, Mestre – FURB
Membro:	Prof. Joyce Martins, Mestre – FURB

Blumenau, 04 de julho de 2016

Dedico este trabalho a todos que contribuíram para a construção de minha vida acadêmica.

AGRADECIMENTOS

À Deus.

À minha família.

Aos meus amigos.

Ao meu orientador.

Aos meus professores.

E Lembre-se: você é seu próprio general. Então, tome agora a iniciativa, planeje e marche decidido para a vitória.

Sun Tzu

RESUMO

Este trabalho apresenta o desenvolvimento de uma ferramenta de programação visual, nomeada Gext JS, com o objetivo de facilitar a construção de páginas *web* baseadas no *framework* Ext JS. A ferramenta é composta por duas partes: um editor gráfico com uma linguagem específica de domínio para a definição dos elementos que compõem as páginas *web*; e um gerador de código, que transforma o modelo construído pelo editor gráfico em código *web*. No desenvolvimento do Gext JS foram utilizadas as ferramentas Eclipse *Modeling Framework* e *Graphical Modeling Framework*, para a construção do editor gráfico, e o motor de transformação Acceleo, para a construção de um gerador de páginas *web* baseado em *templates*. Todas as ferramentas utilizadas são *plugins* da IDE Eclipse, assim como o próprio Gext JS. A principal vantagem de se utilizar o Gext JS é a maior eficiência na construção de páginas *web*, proporcionada tanto pelo editor gráfico quanto pelo gerador de código.

Palavras-chave: Reúso. Geração de código. Linguagem específica de domínio. Ext JS.

ABSTRACT

This document presents the development of a visual tool, called Gext JS, aiming for facilitating the building of web pages based on Ext JS framework. The tool is composed in two parts: a graphic editor with a Domain-Specific Language for the definition of elements composing web pages; and a code generator, which transforms models made in the graphic editor into web code. Gext JS Development has been carried out by using the tools Graphical Modeling Framework, for the construction of the graphic editor, and Acceleo transformation engine, for the construction of a web page generator based on templates. All used tools are plugins for Eclipse IDE, as well Gext JS. The main advantage of using Gext JS is an improvement on the efficiency of web page construction, provided both by graphic editor and code generator.

Key-words: Reuse. Code generation. Domain specific language. Ext JS.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

CSS – *Cascading Style Sheets*

DSL – *Domain-Specific Language*

EAF – *Enterprise Application Frameworks*

EE – *Enterprise Edition*

EMF – Eclipse *Modeling Framework*

Gext JS – Gerador de Ext JS

GMF – *Graphical Modeling Framework*

GPL – *General Purpose Language*

GUI – *Graphical User Interface*

HTML – *HyperText Markup Language*

HTTP – *HyperText Transfer Protocol*

IDE – *Integrated Development Environment*

JPA – Java *Persistence* API

JS – JavaScript

JSF – Java *Server Faces*

JSON – JavaScript *Object Notation*

M2T – *Model to Text*

MDE – *Model-Driven Engineering*

MIF – *Middleware Integration Frameworks*

PHP – *Hypertext Preprocessor*

RF – Requisitos Funcionais

RNF – Requisitos Não Funcionais

SA – Sencha *Architect*

SIF – *System Infrastructure Frameworks*

TI – Tecnologia da Informação

UC – *Use Case*

UI – *User Interface*

UML – *Unified Modeling Language*

URI – *Uniform Resource Identifier*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O software desempenha um papel fundamental em diversas áreas, seja para controle e segurança de informações, controle de processos, apoio a decisões, bem como, entretenimento (HIRAMA, 2011). A sociedade contemporânea não poderia existir sem software, visto que vários setores fazem uso intenso de recursos computacionais. Dessa forma, agilidade, produtividade e qualidade são significantes no desenvolvimento de software (SOMMERVILLE, 2011).

A agilidade e a produtividade podem ser encontradas por meio do reúso de software, caracterizado por ser uma prática que reutiliza artefatos de software preexistentes. Por sua vez, a qualidade do software é comprovada, pois, ao longo do desenvolvimento o reúso proporciona ao desenvolvedor obter códigos já consolidados ao longo da programação. Um exemplo disso é o uso de *frameworks* para apoiar o desenvolvimento de sistemas (BAUER, 2013). Um *framework* consiste de um conjunto de classes cooperativas que implementam os mecanismos essenciais de um domínio específico (HORSTMANN, 2007). Complementando, define-se um *framework* como uma estrutura genérica estendida para se criar uma aplicação (SOMMERVILLE, 2011).

Além dos *frameworks*, a abordagem *Model-Driven Engineering* (MDE), ou Engenharia Dirigida por Modelos, também se baseia na prática de reúso, propondo a construção de software a partir de modelos de diferentes níveis de abstração, sendo que o código fonte é o nível mais inferior (KIRK et al., 2007; SOMMERVILLE, 2011). Os modelos mais abstratos podem ser construídos por meio de uma *Domain-Specific Language* (DSL), ou Linguagem Específica de Domínio, que fornece uma abstração para uma problemática específica. Geradores de código também se baseiam em reúso e podem ser utilizados para ler um modelo criado com uma DSL e gerar o modelo de nível inferior subsequente. Isso é feito, principalmente, para gerar o código fonte do software modelado (VOELTER, 2013).

Diversos *frameworks* são amplamente utilizados na indústria de software (HIRAMA, 2011). Dentre os quais, destaca-se o Ext JS, um *framework* implementado em JavaScript que fornece componentes para a criação de *Graphical User Interface* (GUI), ou Interfaces Gráficas com o Usuário (SENCHA, 2015a). Contudo, apesar do Ext JS auxiliar na criação de GUI, não é possível visualizar a página *web* que está sendo construída até o momento de executar o código. Além disso, utilizar *frameworks* requer do programador um conhecimento detalhado de suas classes e a configuração de diversas variáveis. Devido a essa e outras demandas, surgiu ambiente de desenvolvimento Sencha Architect (SA), que possibilita ao

programador implementar visualmente uma página *web* utilizando o Ext JS, ocultando complexidades relacionadas com o uso desse *framework* (SENCHA, 2015b). Porém, como o Ext JS e o SA são tecnologias pagas para uso comercial, fatores como o custo podem contribuir significativamente para o preço de um software desenvolvido com essas ferramentas. Para eliminar ao menos o custo do SA, uma alternativa seria substituí-lo com a criação de uma outra ferramenta que seja de uso livre. Essa ferramenta poderia ser criada a partir de uma DSL voltada para o Ext JS. Dessa forma, a programação visual das páginas *web* baseadas no Ext JS pode ser feita apenas com o custo de sua licença comercial.

Diante do exposto, foi proposto unir conceitos de MDE, DSL, reúso e geração de código, para criar um editor gráfico que permita a programação visual de páginas *web* que utilizam o *framework* Ext JS. Com esse editor, chamado de Gext JS (Gerador de Ext JS), o desenvolvedor pode criar um modelo da página *web* e gerar o código fonte em Ext JS dessa página.

1.1 OBJETIVOS

O objetivo deste trabalho é criar um editor gráfico que permita a programação visual de páginas *web* baseadas no *framework* Ext JS.

Os objetivos específicos são:

a) criar uma GUI para edição de determinados componentes do Ext JS;
b) disponibilizar um gerador de código fonte que cria o código em Ext JS a partir dos modelos criados na GUI.

1.2 ESTRUTURA

O trabalho está organizado em quatro capítulos: introdução, fundamentação teórica, desenvolvimento da ferramenta e conclusões. O próximo capítulo abrange aspectos teóricos estudados para o desenvolvimento do trabalho. São apresentados temas como *frameworks*, engenharia dirigida por modelos, linguagem específica de domínio e geração de código. Também são relatados alguns trabalhos correlatos de origem acadêmica e comercial. O capítulo 3 aborda o desenvolvimento da ferramenta proposta, englobando os requisitos, a especificação e a implementação. Por fim, o capítulo 4 apresenta as conclusões e as sugestões para possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos fundamentais para o desenvolvimento da ferramenta proposta. Esses conceitos estão organizados da seguinte forma: na seção 2.1, são apresentados detalhes a respeito de *frameworks*; na seção 2.2, aborda-se engenharia dirigida por modelos; na seção 2.3, são descritas as linguagens específicas de domínio; na seção 2.4, são apresentados os conceitos relacionados com geração de código; e na seção 2.5, são apresentados alguns trabalhos correlatos ao trabalho proposto nesta monografia.

2.1 FRAMEWORKS

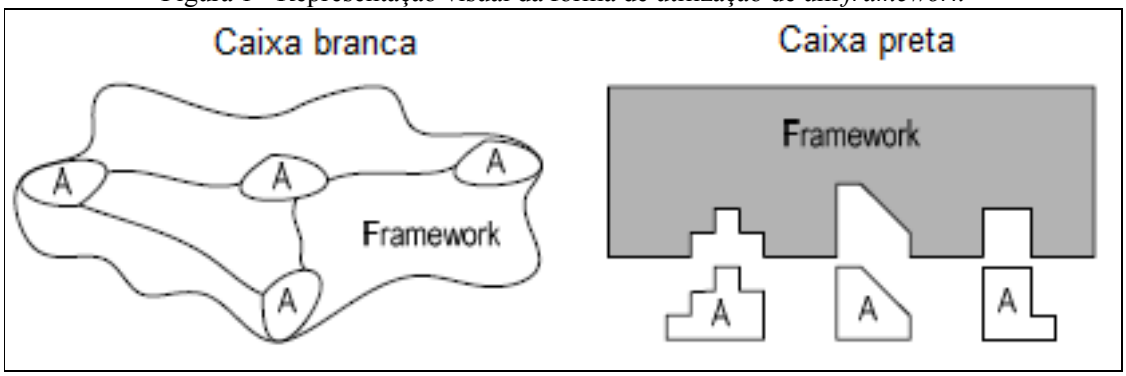
Framework é um artefato de software semi completo que fornece uma construção para aspectos comuns e variáveis de um domínio (FAYAD; JOHNSON, 1999; LOPES et al., 2009). Seu código precisa ser completado para dar origem a uma aplicação específica. Existem *frameworks* que ajudam na implementação de requisitos não funcionais, como persistência de dados, e há aqueles que auxiliam no desenvolvimento de software direcionado a domínios específicos, como do comércio e da indústria (VIANA, 2014).

De acordo com seu propósito os *frameworks* classificam-se em: a) *System Infrastructure Frameworks* (SIF), que auxiliam no desenvolvimento de software que controlam rotinas de baixo nível, como sistemas operacionais, gerenciadores de janela gráfica e software embarcado; b) *Middleware Integration Frameworks* (MIF), que auxiliam na implementação de requisitos não funcionais presentes em muitas aplicações, como persistência de dados e interface *web*; c) *Enterprise Application Frameworks* (EAF), que implementam domínios específicos de negócio (FAYAD; SCHMIDT, 1997). Aplicando esta classificação ao tema do trabalho proposto, verifica-se que o Ext JS é um *framework* do tipo MIF, pois auxilia o desenvolvimento de GUI de sistemas *web*.

Frameworks também podem ser classificados de acordo com a forma de utilização. Neste caso, tem-se: a) caixa branca, quando o uso ocorre por meio de herança de suas classes; b) caixa preta, quando o uso ocorre por meio de criação de objetos e/ou pela chamada dos métodos dos componentes do *framework*; e c) caixa cinza, quando o uso ocorre das duas formas anteriores (ABI-ANTOUN, 2007). De acordo com essa classificação o Ext JS é um *framework* do tipo caixa preta, pois esse *framework* é utilizado instanciando-se objetos de seus componentes. A Figura 1 apresenta a representação do uso de *frameworks* caixa branca à esquerda, e *frameworks* caixa preta à direita, sendo "F" um *framework* e "A" uma aplicação computacional. Nota-se que o *framework* caixa branca está diretamente ligado ao programa que faz seu uso, por se tratar de uma herança de seus componentes. Já o *framework* caixa

preta não possui a mesma dependência, pois o uso é feito por meio de chamadas isoladas (LOPES et al., 2009).

Figura 1 - Representação visual da forma de utilização de um *framework*



Fonte: adaptado de Lopes et al. (2009, p. 258).

Como a utilização do *framework* tipo caixa branca é feita por meio de herança, a construção do software está sujeita aos desafios do uso da herança. Apesar desta oferecer inúmeras vantagens para o polimorfismo de classes, sobrescrever métodos pode quebrar o *design* do *framework*. Sendo assim, esta flexibilidade de adaptação requer um vasto conhecimento sobre como utilizar o *framework*. Na perspectiva de *design* do *framework*, o forte acoplamento de herança torna os *frameworks*, em geral, mais rígidos ou difíceis de evoluir. Isto ocorre devido ao problema da fragilidade da classe base, onde por questões de interface (sintaxe) ou de comportamento (semântica), subitamente quebra-se uma subclasse existente (ABI-ANTOUN, 2007; LOPES et al., 2009).

Já no uso do *framework* tipo caixa preta, o mecanismo de composição é utilizado para a implementação. Assim como no padrão de projeto Estratégia, o *framework* oferece uma interface "A" que deverá ser implementada, para então ser invocada pela classe "B" oferecida também. Desta forma, caso houver um novo comportamento a ser tratado, basta implementar novamente esta interface "A" em outra classe. Este conceito cria uma variabilidade de menor criticidade, pois somente é possível adaptar um comportamento de uma classe. Intrinsecamente, a relação e quantidade de classes e interfaces aumenta. Por outro lado, a implementação de interfaces permite inúmeras extensões, embora, torna-o pouco modificável (ABI-ANTOUN, 2007; LOPES et al., 2009).

Um dos maiores desafios de se utilizar um *framework* é o seu completo entendimento (XU; BUTLER, 2006). Utilizar um *framework* não é uma tarefa trivial, pois demanda um conhecimento detalhado dos seus recursos e como deve ser utilizado em determinadas situações (KIRK et al., 2007). Dessa forma, sua curva de aprendizado é acentuada (SRINIVASAN, 1999). Por exemplo, o programador pode utilizar uma opção de

desenvolvimento genérica que simplifica o trabalho de codificação, ao invés de estender esta opção genérica e implementá-la de acordo com a necessidade. Para simplificar o uso de um *framework*, é necessário considerar a adaptação do *framework* e a definição da aplicação. Adaptação do *framework* consiste em lidar com a computação, fornecendo uma implementação para interfaces que vinculam o *framework* e o programa que o utiliza. Já a definição da aplicação trata da composição, fornecendo instâncias do *framework* e o programa que o utiliza (LOPES et al., 2009).

2.2 ENGENHARIA DIRIGIDA POR MODELOS

O termo Engenharia Dirigida por Modelos, ou *Model-Driven Engineering* (MDE), é tipicamente usado para descrever abordagens em que o desenvolvimento de software ocorre em modelos de diferentes níveis de abstração, sendo que o código fonte é considerado o modelo de nível mais baixo (FRANCE; RUMPE, 2007). Cada modelo é gerado a partir do modelo de nível anterior. Por exemplo, ao modelar uma entidade Pessoa, pode-se modelar com um alto nível de abstração, definindo características, como cor dos olhos, idade e cor do cabelo. Em um outro modelo com um nível mais baixo de abstração, específico para a linguagem Java, por exemplo é especificado o tipo de dado (`String`, `Integer`, `String`, respectivamente) dessas características. A partir desse último modelo, pode ser gerado o código na linguagem Java. Dessa forma, o projetista de software preocupa-se com o domínio do sistema e não com aspectos de implementação e plataforma (SOMMERVILLE, 2011).

A infraestrutura da MDE necessita de um propósito para a criação de modelos de interesse e um propósito para a transformação destes modelos em um software concreto. Como exemplo, destaca-se a *Unified Modeling Language* (UML) como uma das linguagens de modelagem mais utilizadas na indústria de software, fornecendo abstrações complexas para entidades, relacionamentos, estereótipos, entre outros. Estes relacionamentos são então transformados e codificados, em grande parte, manualmente (SOMMERVILLE, 2011).

A MDE proporciona uma facilidade maior de entendimento para o ser humano, algo que as linguagens de programação da terceira geração tentam atingir. Este entendimento é auxiliado pela forma em que o uso da MDE ocorre, pois divide-se em:

a) Linguagem de modelagem de domínio específico ou DSLs: possui um sistema de tipos que formaliza a estrutura da aplicação, o comportamento e os requerimentos dentro de domínios particulares, como é o caso de aplicações bancárias, ou sistemas de tráfegos, ou até mesmo componentes gráficos. Desenvolvedores

estruturam a futura aplicação por meio de metamodelos, de forma declarativa, ao invés de imperativamente (SOMMERVILLE, 2011);

b) Motores de Transformação e/ou geradores de código: são responsáveis por analisar os modelos e sintetizar seus relacionamentos. Esta sintetização garante a consistência entre implementação da aplicação e a qualidade do serviço desenvolvido. Este processo é comumente chamado de "correção por construção" (SOMMERVILLE, 2011).

Possibilitar que desenvolvedores construam um sistema a partir de um modelo de maior nível de abstração reduz a probabilidade de erros, acelera o processo de projeto e implementação e permite a criação de modelos reusáveis independentes da plataforma. Por meio de ferramentas específicas, pode-se gerar o código para diferentes linguagens a partir do mesmo modelo. Sendo assim, caso seja necessário alterar a linguagem da aplicação, basta alterar o gerador de código deste modelo (SOMMERVILLE, 2011).

Por outro lado, a MDE também apresenta desvantagens. Conforme explanado anteriormente, modelos são uma boa maneira de abordar o projeto de um sistema, porém, as abstrações promovidas pelos modelos nem sempre são completamente corretas para a implementação. Desta forma, pode-se criar modelos informais do projeto. Outra desvantagem refere-se ao fato de que os programas gerados podem não ser tão eficientes quanto os desenvolvidos manualmente, pois o gerador de código usa implementações genéricas que podem não ser específicas para determinada situação (SOMMERVILLE, 2011). A rede de modelos mantida em um ambiente MDE é um reflexo da complexidade inerente do software. Ou seja, a criação de modelos de software trabalha com a subjetividade e podem surgir modelos complexos para soluções simples, devido à complexidade de se aprender linguagens de modelagem. Os desafios atuais para utilizar a MDE refletem que esta é uma área em ascensão e pode ser explorada e melhorada continuamente (FRANCE; RUMPE, 2007).

2.3 LINGUAGEM ESPECÍFICA DE DOMÍNIO

Uma Linguagem Específica de Domínio, ou *Domain-Specific Language* (DSL), tem como objetivo fornecer uma abstração para um problema específico. Comparada à uma Linguagem de Propósito Geral, ou *General Purpose Language* (GPL), como a *Unified Modeling Language* (UML), uma DSL tem elementos e relacionamentos específicos para o seu domínio, seu tamanho tende a ser menor e sua evolução é facilitada por conta de seu escopo ser reduzido (VOELTER, 2013). Uma DSL é um meio de descrever e gerar membros de uma família de programas no domínio modelado (DEURSEN; KLINT, 2002).

DSLs podem ser utilizadas para solucionar parte dos desafios encontrados no uso da MDE (FRANCE; RUMPE, 2007), como é o caso da modelagem de um programa. Em termos de implementação, a DSL é dividida em sintaxe abstrata e sintaxe concreta (VOELTER, 2013). Na sintaxe abstrata são definidas as características dos elementos e os relacionamentos que compõem o domínio modelado. A sintaxe abstrata tem o propósito de ser o núcleo da DSL, sendo responsável por validações e transformações para a geração de código. Existem diversas ferramentas que auxiliam o desenvolvimento da DSL para definir a sintaxe abstrata (VOELTER, 2013), dentre as quais destaca-se a versão do Ambiente de Desenvolvimento Integrado, ou *Integrated Development Environment* (IDE), Eclipse chamada *Eclipse Modeling Framework* (EMF).

Na sintaxe concreta é possível definir os elementos que interagem diretamente com o utilizador da DSL, que foram definidos na sintaxe abstrata. Esses elementos podem ser de notação textual (o programador escreve a linguagem definida na DSL), de notação gráfica (o programador utiliza componentes gráficos que podem ser manipulados) e de notação tabular (o programador utiliza uma tabela que representa a DSL) (VOELTER, 2013). Como exemplo de ferramenta para definir a sintaxe concreta existe o *plugin* da IDE Eclipse chamado *Graphical Modeling Framework* (GMF) (GRONBACK, 2009; THE ECLIPSE FOUNDATION, 2015b).

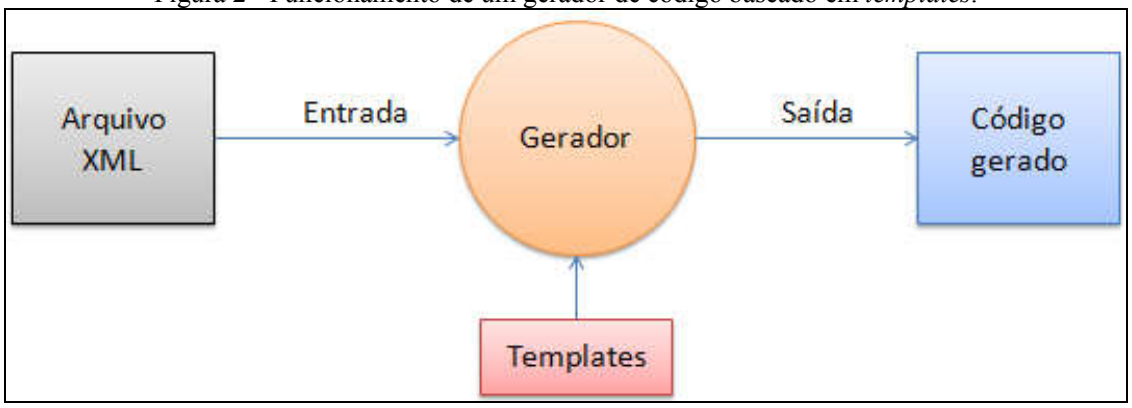
2.4 GERAÇÃO DE CÓDIGO

Geração de código trata de desenvolver programas que escrevem programas (HERRINGTON, 2003). A geração de código é uma técnica que utiliza ferramentas específicas para gerar programas de computador. Tais ferramentas podem variar desde *scripts* de ajuda simples até aplicações completas (SILVEIRA, 2006).

Geradores de código distinguem-se em duas categorias: passiva e ativa. No modelo passivo o gerador não se responsabiliza pelo código após sua geração, delegando esta responsabilidade para o desenvolvedor. No modelo ativo o gerador mantém a responsabilidade a longo prazo da aplicação, assim como grande parte da codificação. Para o modelo ativo, as mudanças necessárias na aplicação são realizadas pelo gerador de código, sendo necessário executá-lo com frequência (HERRINGTON, 2003). Um exemplo de geração de código passivo é o *wizard* da IDE Eclipse, que pode criar uma simples classe em Java. Para a geração de código ativa, tem-se o recurso de construção de interfaces gráficas Java da IDE NetBeans (ORACLE, 2015) e a ferramenta Genexus, que podem gerar e manter uma aplicação completa (ARTECH CONSULTORES SRL, 2015).

Um gerador de código é um motor de transformação que funciona de forma semelhante ao de uma função de uma linguagem de programação. Ele recebe como entrada os dados sobre o conteúdo a ser gerado, realiza um processamento e retorna como saída o código resultante (ALMEIDA et al., 2009). É necessário que o formato de entrada seja estruturado, por exemplo, como um arquivo *eXtensible Markup Language* (XML), para facilitar a extração dos dados por parte do gerador. Em alguns geradores, o formato dos artefatos gerados está embutido no código do próprio gerador. A saída desses geradores é sempre o mesmo tipo de arquivo, variando o conteúdo com base na entrada recebida. Porém, existem geradores que são configuráveis e o formato dos artefatos gerados é definido por meio de *templates* (HERRINGTON, 2003). Um *template* é um modelo a ser seguido, que possui uma estrutura predefinida e permite a automatização da criação de novos conteúdos a partir desse modelo (HERRINGTON, 2003; VOELTER, 2013). A Figura 2 ilustra o funcionamento de um gerador de código baseado em *templates*.

Figura 2 - Funcionamento de um gerador de código baseado em *templates*.



2.5 TRABALHOS CORRELATOS

Nesta seção são abordados três trabalhos correlatos. A seção 2.5.1 trata da ferramenta comercial Sencha *Architect*. A seção 2.5.2 apresenta outra ferramenta comercial chamada Genexus e por fim, a seção 2.5.3 exibe o trabalho de conclusão de curso de Battisti (2014), nomeado Elicitar.

2.5.1 Sencha *Architect*

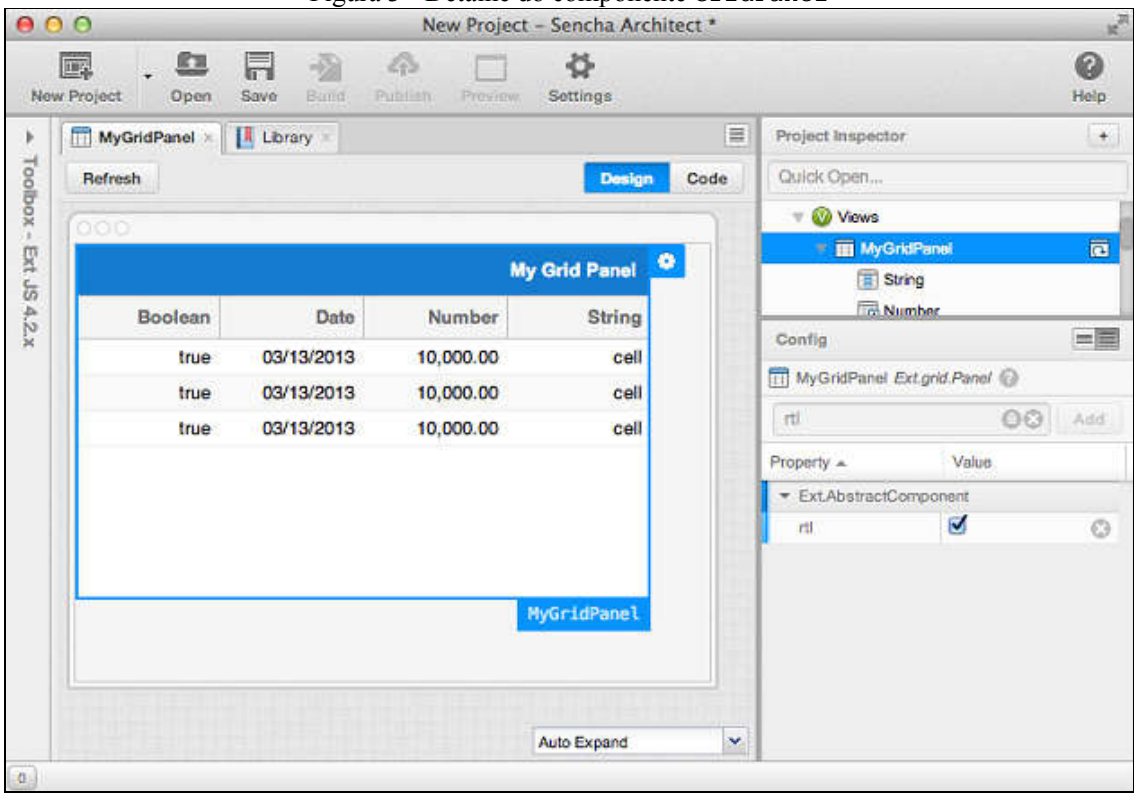
Substituto do antigo Ext *Desing*, o Sencha *Architect* (SA) é uma ferramenta que possibilita criar aplicativos para a *web* e para dispositivos móveis. Pertencente à empresa Sencha, o SA é um ambiente de desenvolvimento de licença comercial paga sendo um aplicativo *desktop*, disponibilizado nas plataformas Windows, Mac e Linux. Conta com

diversos *templates* predefinidos para auxiliar o desenvolvedor, sendo possível inclusive customizar de forma dinâmica todo o *layout* e o relacionamento entre os componentes GUI de acordo com a necessidade (SENCHA, 2015b).

Com o conceito *drag and drop*, é uma alternativa para criar aplicativos visualmente. O código gerado é em JavaScript, mais precisamente em outro *framework* da Sencha, o Ext JS. O desenvolvedor pode optar entre a perspectiva visual e a textual para codificar a aplicação manualmente (SENCHA, 2015b).

O SA é o parâmetro de estudo para desenvolvimento do trabalho proposto. Na Figura 3 é mostrada a edição do componente `GridPanel`, que exibe para o usuário final uma tabela com linhas e colunas com dados.

Figura 3 - Detalhe do componente `GridPanel`

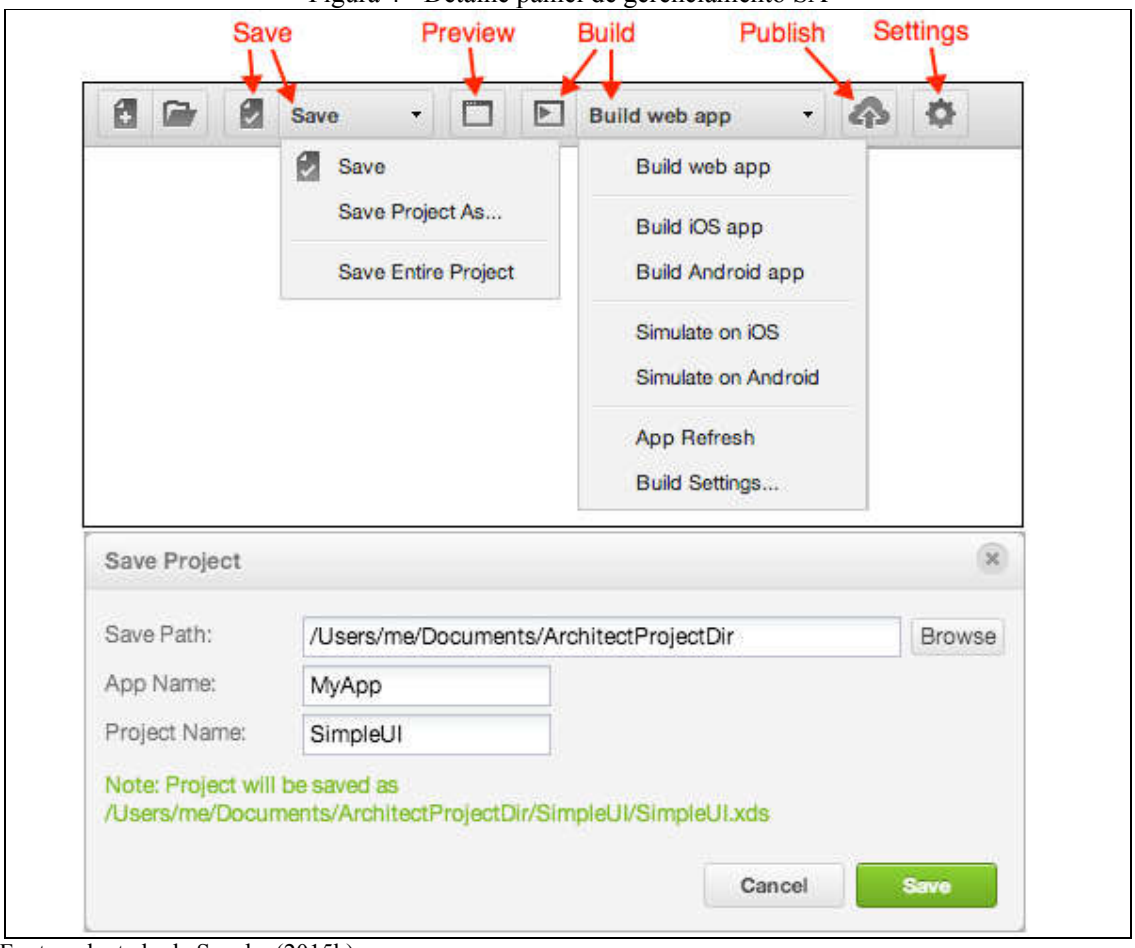


Fonte: Sencha (2013b).

O SA conta com um recurso nativo para controle de versão. Com ele é possível compartilhar os projetos com o time de desenvolvimento. Existem outros recursos que facilitam o gerenciamento do desenvolvimento como por exemplo o `Preview`, o `Simulate` e o `Build`. O `Preview` possibilita ao desenvolvedor executar o projeto diretamente no *browser*. O `Simulate` fornece uma simulação de um dispositivo móvel para testar as aplicações para *smartphones*, sem a necessidade de instalar o software em um aparelho físico. Essa opção fornece suporte para as plataformas Android e iOS. Já o `Build` compacta a aplicação para

distribuição comercial, para o Google Play e a Apple Store. Conforme Figura 4, é possível acompanhar algumas dessas funcionalidades entre outras, por meio do painel de administração do SA.

Figura 4 - Detalhe painel de gerenciamento SA

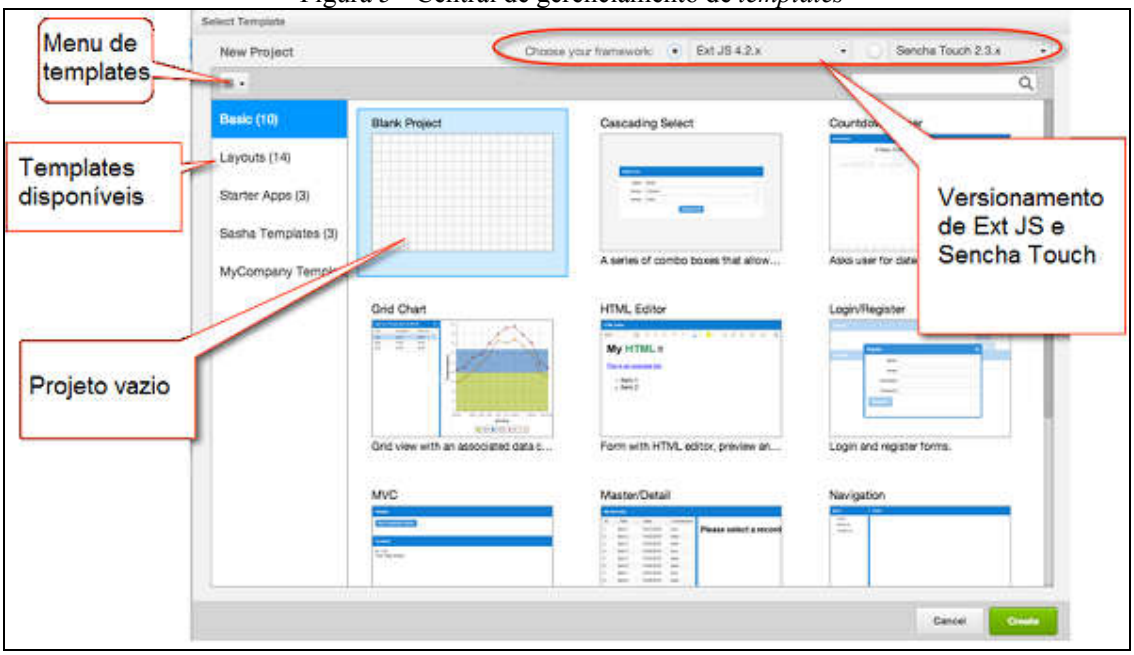


Fonte: adaptado de Sencha (2015b) .

Para a utilização destes recursos ser possível, é necessário instalar as tecnologias Ruby, para processamento de CSS, Java e Apache Ant, utilizado para o *deploy* de projeto Android. Todos estes softwares secundários podem ser instalados por meio da central de *downloads* que a próprio SA fornece. O SA também fornece nativamente um servidor *web* próprio para testes das aplicações, além de possibilitar a instalação de outros servidores *web*.

O SA possui uma grande gama de *templates* predefinidos que podem ser utilizados ou também estendidos. A Figura 5 exibe a central de gerenciamento de *templates*. Nela pode-se observar que é possível simular diferentes versões do Ext JS, assim como utilizar *templates* vazios ou com opções mais completas, como é o caso de formulário de autenticação. Estes recursos viabilizam o desenvolvimento para diferentes versões do Ext JS.

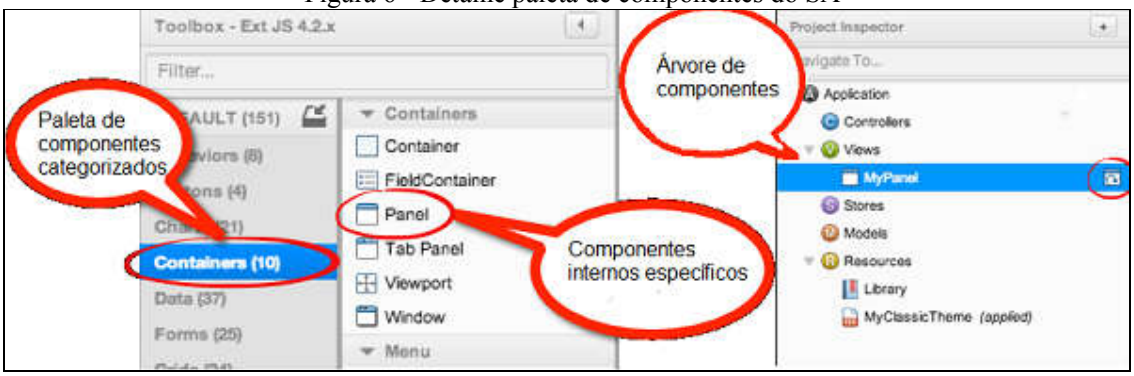
Figura 5 - Central de gerenciamento de *templates*



Fonte: adaptado de Sencha (2015b) .

Outro recurso que facilita o uso do SA é a sua interface gráfica organizada. Conforme Figura 6. Observa-se que os componentes são categorizados com paletas em forma de árvore, além de possuir um filtro para busca dos componentes disponíveis.

Figura 6 - Detalhe paleta de componentes do SA



Fonte: adaptado de Sencha (2015b) .

Sem dúvidas a IDE de desenvolvimento SA cumpre o papel de proporcionar um bom ambiente de desenvolvimento para o *framework* Ext JS. Por se tratar do mesmo fornecedor da linguagem e da IDE, certamente existe eficiência no código gerado.

2.5.2 Genexus

A ferramenta Genexus surgiu na década de 80, após integrantes da empresa Artech do Uruguai se depararem com problemas comuns no desenvolvimento de software de grandes

aplicações corporativas. A cada novo desenvolvimento de aplicação encontravam-se as mesmas problemáticas, como análise de objetivos, conhecimento técnico, análise e desenvolvimento de um modelo que corresponda com a realidade (CHAVES, 2003). Seu objetivo é permitir desenvolver e manter aplicativos corporativos para múltiplas plataformas, automatizando a geração de toda a aplicação (ARTECH CONSULTORES SRL, 2015).

A geração da aplicação é baseada em modelos que o desenvolvedor cria para abstrair uma determinada funcionalidade do sistema. Esses modelos estão diretamente relacionados com as tabelas do banco de dados, que criam por exemplo, uma página em *HyperText Markup Language* (HTML) para interação com o usuário final. Tais plataformas incluem conteúdo para *desktop*, *web* e dispositivos móveis, assim como o banco de dados da aplicação (ARTECH CONSULTORES SRL, 2015).

O desenvolvimento do sistema é modular e por protótipos. Como são abstraídas várias etapas da programação e os requisitos não funcionais já estão estabelecidos, o Genexus possibilita que os analistas de sistemas e/ou desenvolvedores implementem aplicações em menor tempo e com maior robustez, além de auxiliar os usuários ao longo do ciclo de vida do sistema (CHAVES, 2003).

O Genexus não se limita a somente gerar um trecho isolado de código de uma aplicação, mas sim disponibilizar toda a aplicação em várias linguagens para várias plataformas, sendo necessário apenas criar e prestar manutenção ao modelo da aplicação (ARTECH CONSULTORES SRL, 2015). O Genexus está classificado como um gerador de código ativo. A abordagem de desenvolvimento em Genexus está diretamente ligada à concepção de um sistema por meio de modelos, logo, faz parte da MDE.

2.5.3 Elicitar

Battisti (2014) desenvolveu a ferramenta Elicitar, um protótipo de gerador de código a partir de especificações de padrões de requisitos, escritos na língua portuguesa. A ferramenta é capaz de processar requisitos baseados em padrões de requisitos de informações, processar requisitos escritos em língua portuguesa com vocabulário irrestrito, processar componentes visuais de interface gráfica, gerar arquivos de definição JavaScript *Object Notation* (JSON) e/ou XML que corresponde à interface gráfica descrita e gerar código a partir do arquivo de definição na linguagem Java *Server Faces* (JSF).

O processo de geração das páginas HTML se divide em quatro etapas: cadastrar os requisitos de acordo com o padrão de requisitos, utilizar processamento da linguagem natural

para elencar os componentes da interface, gerar o arquivo de definição (em XML e/ou JSON) e gerar o código HTML a partir deste arquivo de definição.

O produto final gerado é uma aplicação *web* que une as tecnologias Java *Persistence* API (JPA) para prover o mapeamento objeto relacional no banco de dados, JSF para interação com o usuário final, utilizando o *framework* PrimeFaces. As tecnologias para geração de código englobaram o motor de *templates* Velocity, para geração de páginas HTML, e a biblioteca CoGroo, para análise léxica morfológica, ou seja, verificação da estrutura de uma frase organizando suas palavras em artigo, verbo, preposição entre outras categorias gramaticais.

A implementação dividiu-se em associação entre os requisitos, extração das características de um requisito, definição dos componentes de interface gráfica a partir das características, geração do arquivo de definição e geração de código HTML a partir do arquivo de definição. Já as etapas de uso do protótipo seguem os passos de cadastrar de um requisito, gerar o arquivo de definição e, por fim, gerar o HTML. É possível ainda aumentar o dicionário interno de características.

A tela principal do protótipo Elicitar é exibida na Figura 7. Nela pode-se verificar as principais funcionalidades que são:

a) `cadastrar requisito`: responsável por discriminar o tipo, que representa o padrão do requisito, o nome, que representa o nome do requisito, o objetivo, que representa uma descrição do propósito deste requisito e a descrição, que descreve as informações do requisito;
b) `gerar definição`: responsável por criar o arquivo em XML e/ou JSON que armazena o conteúdo do requisito;
c) `gerar HTML`: responsável por criar o código fonte em HTML por meio do arquivo de definição;
d) `cadastrar característica`: responsável por especificar a definição do tipo do dado que será incluso no formulário. Por exemplo, definir que um atributo idade somente aceitará valores numéricos.

Os demais ícones não estão relacionados com o foco principal da ferramenta, sendo assim, são opções secundárias. Em especial, a funcionalidade de cadastrar um requisito permite criar uma série de propriedades para um requisito. Após o requisito ser cadastrado, segue-se com processo de geração do arquivo de definição. Esse arquivo é criado por meio do botão `Gerar definição`. A geração da página é feita por meio da opção `Gerar HTML`. Ao

pressionar esse botão deve-se selecionar o arquivo de definição gerado anteriormente. Após esse processo, a página é gerada a partir dos requisitos cadastrados.

Figura 7 - Tela principal do protótipo

Fonte: Battisti (2014, p. 42).

3 DESENVOLVIMENTO DA FERRAMENTA

Neste capítulo tem-se uma descrição do desenvolvimento da ferramenta proposta. Na seção 3.1, são apresentados os requisitos que definiram a funcionalidade da ferramenta. Na seção 3.2, é apresentada a especificação da ferramenta com o auxílio de diagramas da UML. Na seção 3.3, é apresentada a implementação da ferramenta, incluindo as técnicas e ferramentas utilizadas, a criação do editor gráfico, a criação do gerador de código, a operacionalidade da ferramenta e por fim, os resultados obtidos.

3.1 REQUISITOS

O Gext JS é um gerador de código em JavaScript voltado para o *framework* Ext JS. Por meio de um modelo abstrato da página, criado com o auxílio de uma interface gráfica, o código fonte é gerado. Além da interação gráfica com o usuário, o Gext JS possui opções padrões para a geração de código, desta forma o usuário da ferramenta não necessita configurar todas as propriedades obrigatórias de uma página *web*.

As páginas *web* devem seguir um padrão do tipo `Tag` ou `Page`. A `Tag` é uma página que define a estrutura básica de uma interface *web* e pode ser reutilizada por outras páginas. Uma `Tag` normalmente contém um menu de navegação, *layout* de borda de outras páginas *web* e locais específicos para adicionar o conteúdo de outras páginas. Já uma `Page` pode reutilizar uma `Tag` para definir seu *layout*. `Page` também pode conter componentes de formulário de páginas *web*.

Após a modelagem de uma `Tag` e/ou `Page`, deve ser possível a validação dos modelos criados. Caso a validação retorne um erro, uma mensagem resumida o descreve, informando o erro e o modelo a ser corrigido. Após a validação, a ferramenta deve permitir a geração de código fonte desses modelos via a criação de um arquivo. O nome desse arquivo é customizado pelo usuário.

Os Requisitos Funcionais (RF) e Não Funcionais (RNF) atendidos pelo Gext JS, são listados respectivamente no Quadro 1 e no Quadro 2.

Quadro 1 - Requisitos funcionais

Requisitos funcionais(RF)
RF01: permitir a modelagem da GUI de sistemas *web* com base nos componentes do Ext JS, disponíveis na ferramenta.
RF02: permitir a validação dos modelos criados.
RF03: permitir a geração de código em Ext JS a partir dos modelos criados na ferramenta.

Quadro 2 - Requisitos não funcionais

Requisitos não funcionais(RNF)
RNF01: ser um *plugin* voltado para a IDE Eclipse.
RNF02: ter a DSL para modelagem de GUI de sistemas *web* construída a partir dos *plugins* EMF e GMF da IDE Eclipse.
RNF03: ter o gerador de código construído a partir do *plugin* Acceleo da IDE Eclipse.

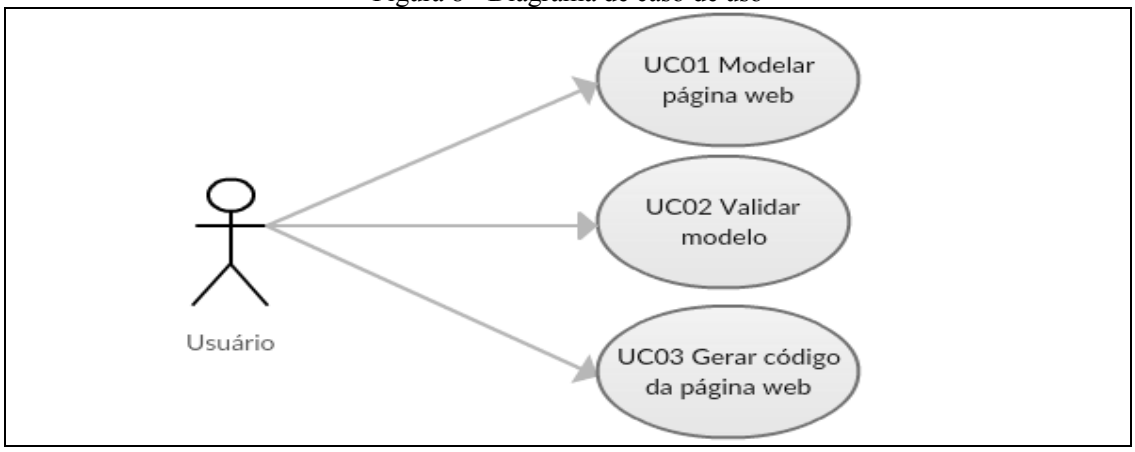
3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação do Gext JS. Primeiramente, é apresentado um diagrama de casos de uso da UML, que resume as ações que os usuários podem realizar com o apoio da ferramenta. Em seguida, é mostrado um diagrama de atividades que apresenta maiores detalhes dessas ações. Em seguida, é apresentado o diagrama de componentes que exibe a divisão interna da ferramenta. Por fim, é mostrada a relação de componentes e suas propriedades contidos no Gext JS.

3.2.1 Diagrama de casos de uso

As ações possíveis para o usuário são apresentadas no diagrama de casos de uso da Figura 8. O usuário pode modelar a página *web*, validar o modelo criado e gerar o código fonte. O usuário pode validar o modelo criado para verificar se algum modelo (ou concretamente um componente) está inconsistente (por exemplo, criar um `Combobox` sem conteúdo). Por fim, o usuário deve gerar o código da página *web* a partir do modelo criado.

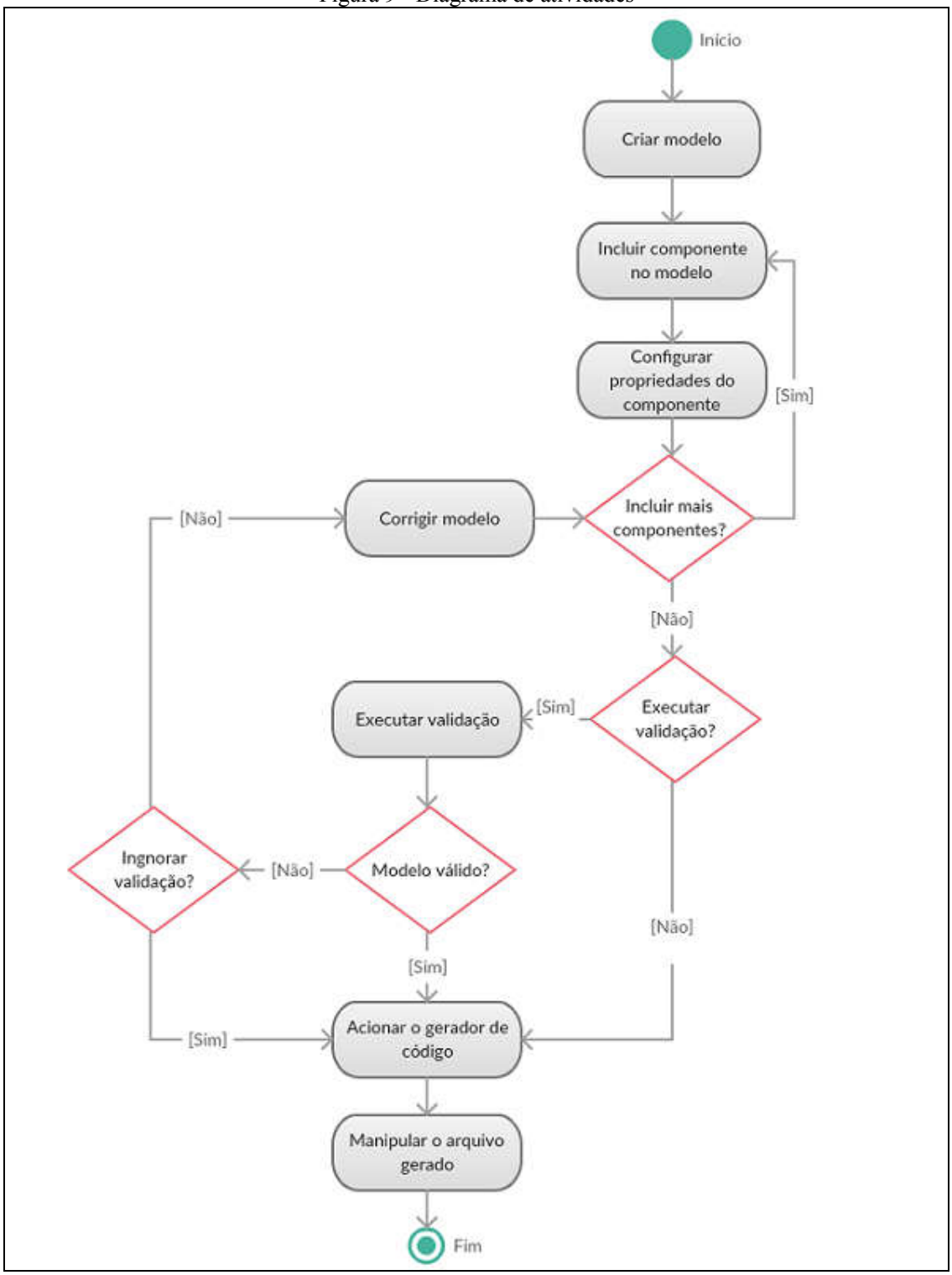
Figura 8 - Diagrama de caso de uso



3.2.2 Diagrama de atividades

Na Figura 9 é apresentado um diagrama de atividades que ilustra a utilização do Gext JS. O usuário pode criar um novo modelo de página *web*. A página *web* possui vários componentes, esses componentes serão componentes gráficos da página. Cada componente possui suas propriedades e elas devem ser configuradas após inclusão do mesmo no editor gráfico. Após a modelagem, recomenda-se que o usuário realize uma validação no modelo (opção *Validate*) antes de iniciar a geração do código da página *web*. Caso essa ação seja ignorada e o modelo contenha informações inválidas ou esteja incompleto, a geração de código possivelmente gerará uma página *web* com defeitos.

Figura 9 - Diagrama de atividades



3.2.3 Relação de modelos e suas propriedades

A Figura 10, Figura 11 e Figura 12 apresentam todos os modelos criados baseados nos componentes do Ext JS. Dentre eles destacam-se:

a) `Model`: elemento raiz que representa o espaço no qual o usuário modela as páginas *web*. É criado automaticamente pelo *wizard* do Eclipse e é o primeiro componente da árvore de componentes;
b) `AbstractPage`: abstração que reúne as informações comuns para `Page` e `Tag`;
c) `Tag`: representa uma página *template*, que pode ser reutilizada em várias páginas *web*. Nela são definidos o *layout* padrão `BorderLayout` da página e os itens de navegação contidos no `Toolbar`;
d) `Toolbar`: componente gráfico que armazena um único `Menu`;
e) `Menu`: componente gráfico que armazena vários `MenuItem`;
f) `MenuItem`: componente gráfico responsável por realizar redirecionamentos para outras páginas *web*;
g) `Page`: uma classe concreta de `AbstractPage`. Nela são incluídos os componentes concretos `Window`, `Panel` e um único `Gridpanel`;
h) `Window`/`Panel`: uma janela e um painel, respectivamente. Em ambos pode-se incluir componentes do tipo `AbstractComponent`;
i) `AbstractComponent`: agrupa várias propriedades comuns dos componentes de formulário, como campos de texto, rótulos, botões, entre outros. Cada componente possui sua particularidade, mas em geral o atributo `id` está presente;
j) `Label`/`Combobox`/`Textfield` e demais componentes de formulário: classes concretas de `Component`. Cada componente possui sua própria finalidade. O `Label` exibe um texto, o `Combobox` exibe uma caixa de seleção única, o `Textfield` é um campo de entrada de texto, e os demais componentes são comuns em formulários de interação com o usuário na área de TI;
k) `Store`: estrutura que armazena dados no formato JSON para serem incluídos em algum componente que necessite apresentar e/ou manipular dados;
l) `SFrontE`: estrutura concreta de `Store` que faz a requisição dos dados para o servidor *back-end*;
m) `SBackE`: estrutura concreta de `Store` que é instanciado com os dados em seu construtor, desta maneira os dados são fornecidos pelo cliente *front-end*;
n) `Gridpanel`: estrutura do tipo tabela que armazena os dados da `Store`;
o) `Abstractcolumn`: colunas que são inseridas no `Gridpanel`;
p) `Column`: coluna do `Gridpanel` para simples apresentação dos dados;
q) `Actioncolumn`: definição de uma coluna especial responsável por apresentar uma

figura na célula e adicionar um evento de clique.

Figura 10 - Relação de modelos de componentes parte 1

Figura 11 - Relação de modelos de componentes parte 2

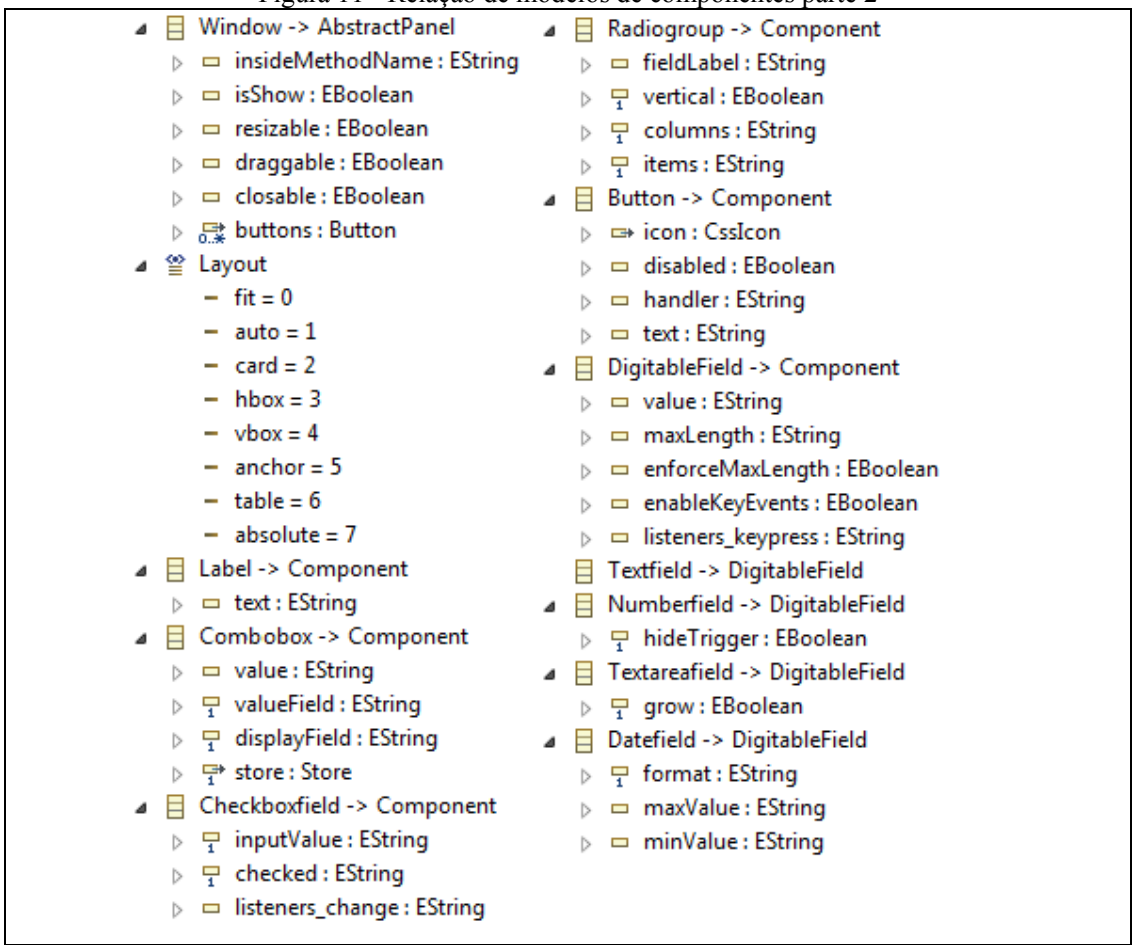
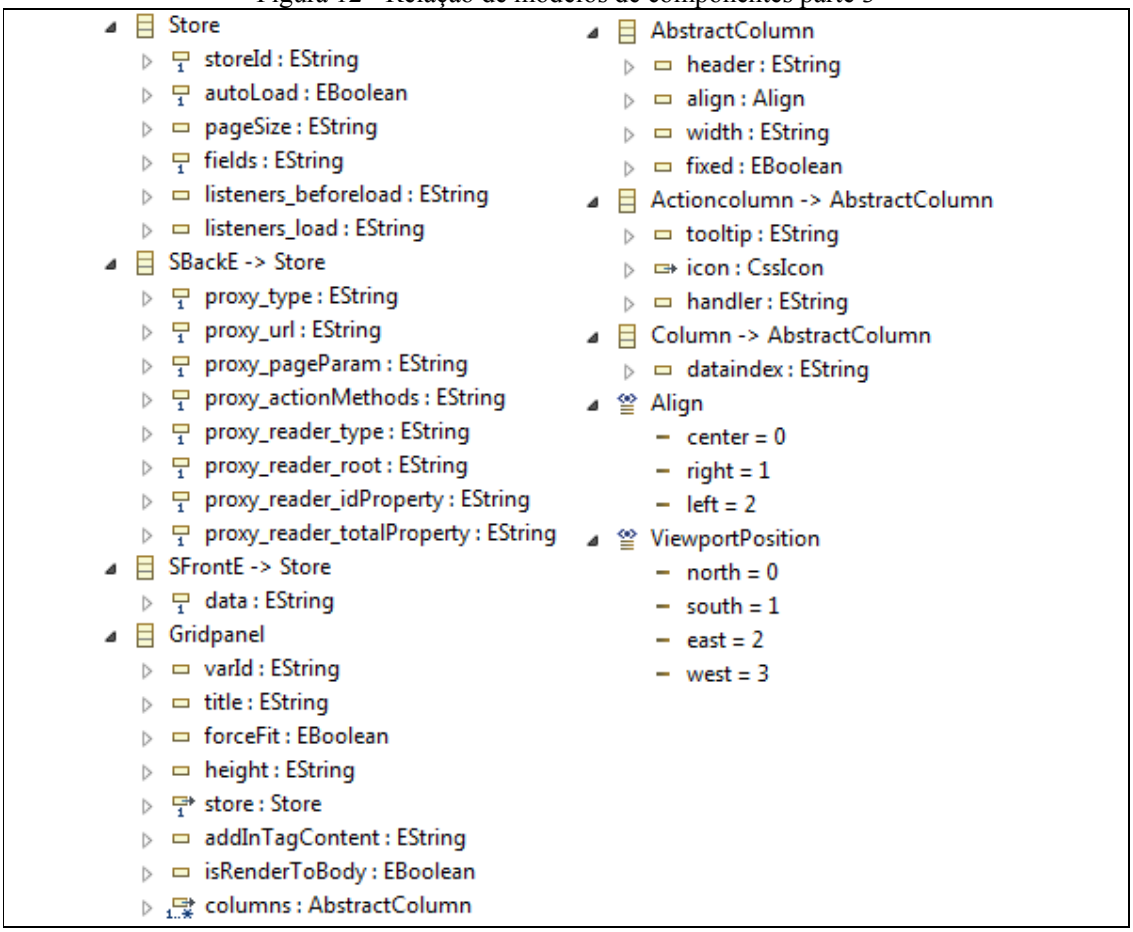


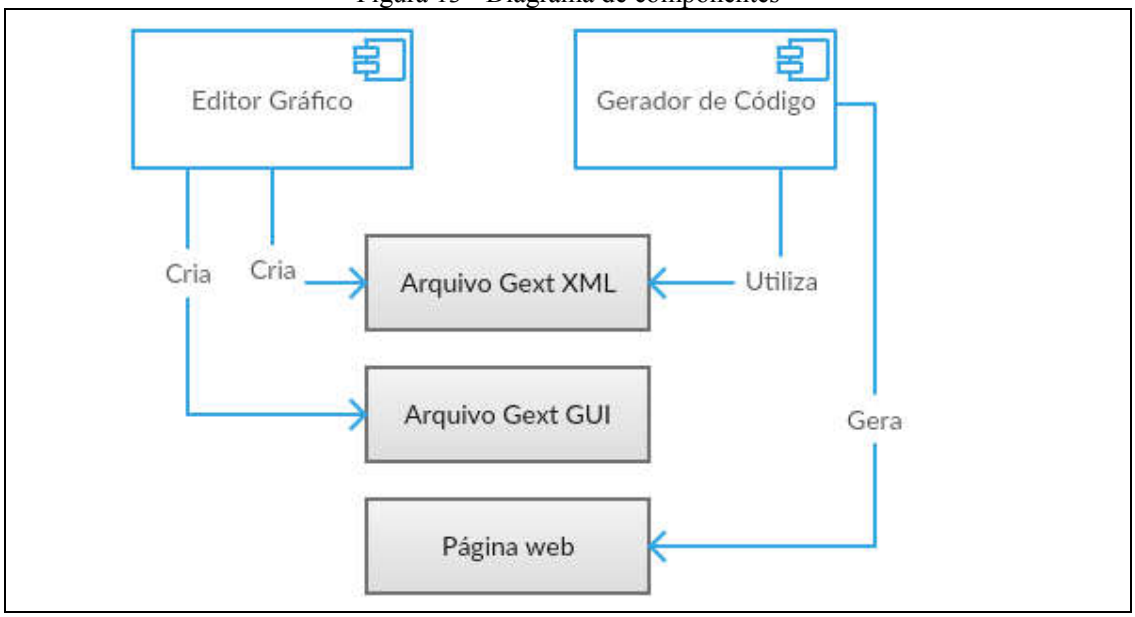
Figura 12 - Relação de modelos de componentes parte 3



3.2.4 Diagrama de componentes

Na Figura 13 é mostrado o diagrama de componentes do Gext JS. Nela é possível perceber que a ferramenta é composta de dois componentes: o editor gráfico e o gerador de código das páginas *web*. O editor gráfico é usado para modelar as páginas *web*, que origina dois artefatos do Gext. O primeiro artefato é um arquivo XML que contém somente os dados necessários para a geração de código. O segundo artefato é um arquivo da GUI que contém informações úteis para o editor gráfico e não é utilizado na geração de código. O gerador de código lê os dados da página modelada, contidos no primeiro artefato criado pelo editor, e constrói o código fonte correspondente.

Figura 13 - Diagrama de componentes



3.3 IMPLEMENTAÇÃO

Nesta seção são descritas as técnicas e ferramentas utilizadas para a implementação do Gext JS.

3.3.1 Técnicas e ferramentas utilizadas

As ferramentas utilizadas para o desenvolvimento do Gext JS foram as seguintes:

a) Eclipse *Modeling Tools*, versão Mars.2 4.5.2: ambiente de desenvolvimento utilizado para construção do Gext JS;

b) *Graphical Modeling Framework* (GMF): *plugin* presente no Eclipse *Modeling Tools* utilizado para construção do editor gráfico do Gext JS;

c) *Eclipse Modeling Framework* (EMF): *plugin* presente no Eclipse *Modeling Tools* utilizado para construção do editor gráfico do Gext JS;

d) Acceleo: *plugin* do Eclipse *Modeling Tools* que permite a criação de geradores de código com base em *templates*;

e) Eclipse Java EE IDE *for Web Developers*, versão Mars 4.5.0: utilizado na criação das páginas *web* com o Ext JS que serviram de modelo para construção dos *templates* do gerador de código do Gext JS;

f) *Framework* Ext JS versão 4.0.7: utilizado na construção das páginas *web*.

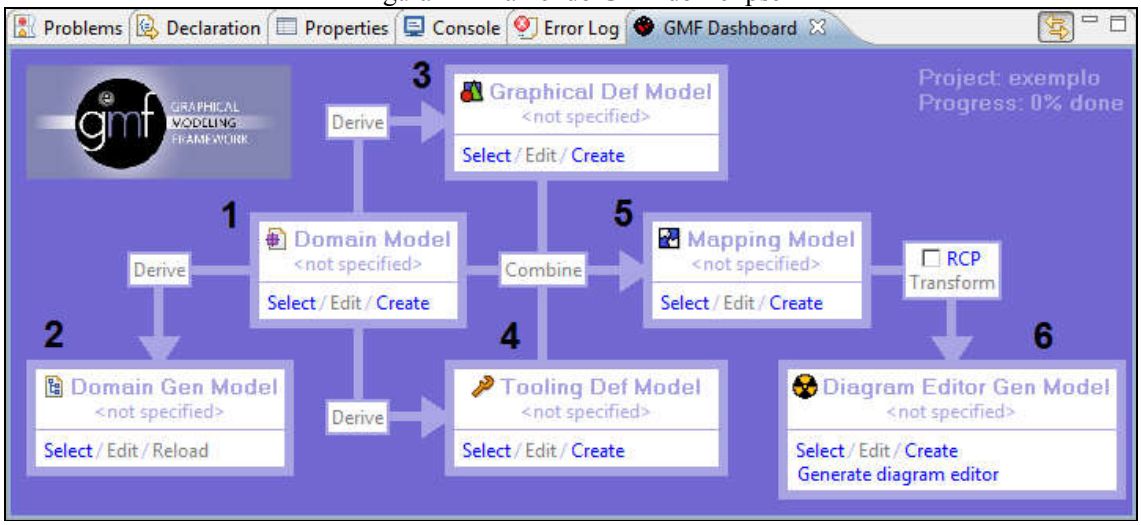
O desenvolvimento do Gext JS foi realizado na seguinte ordem: primeiramente, foi construído o editor gráfico utilizando o EMF e o GMF, então, foi construído o gerador de código utilizando o Acceleo. As seções 3.3.1.1 e 3.3.1.2 descrevem cada uma dessas partes.

3.3.1.1 Construção do editor gráfico do Gext JS

A modelagem das páginas *web* é feita com base em uma DSL. O editor gráfico do Gext JS é a implementação dessa DSL. Para isso, foram utilizado o GMF e o EMF, que permitem a construção de DSLs a partir de um conjunto de modelos. Na Figura 14 é mostrado um *print screen* do GMF *Dashboard*, que é um painel presente no Eclipse IDE que auxilia na construção dos modelos do GMF e o EMF. Esses modelos são:

a) *Domain Model*: define a sintaxe abstrata da DSL, ou seja, a definição dos elementos que fazem parte da linguagem;
b) *Domain Gen Model*: é um modelo gerado pelo Eclipse a partir do *Domain Model* e é responsável pelos *plugins* de definição da DSL do editor gráfico;
c) *Graphical Definition Model*: define a sintaxe concreta da DSL, que no caso do Gext JS, corresponde à aparência gráfica dos elementos do editor;
d) *Tooling Definition Model*: a partir do qual se cria o painel (paleta de componentes) de seleção de elementos do editor gráfico;
e) *Mapping Model*: realiza a vinculação entre os elementos do *Domain Model* com os elementos do *Graphical Definition Model* e do *Tooling Definition Model*;
f) *Diagram Editor Gen Model*: é gerado pelo EMF e GMF a partir do *Mapping Model* para gerar o *plugin* do painel do editor gráfico (THE ECLIPSE FOUNDATION, 2015b).

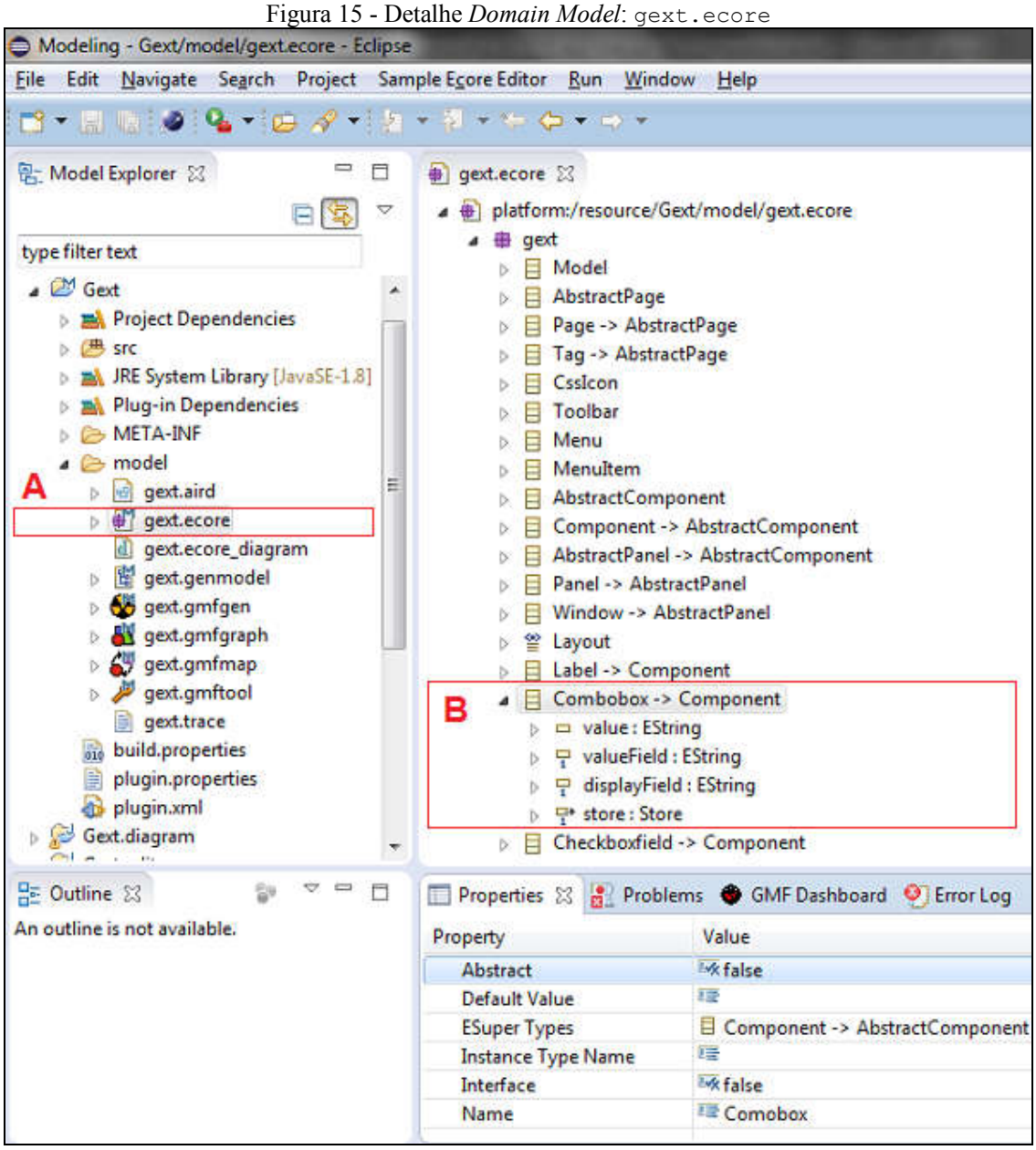
Figura 14 - Painel do GMF do Eclipse



O primeiro modelo criado para a construção do editor gráfico do Gext JS foi o *Domain Model*. Nesse modelo foram implementados os principais elementos gráficos utilizados em páginas *web* construídas com o *framework* Ext JS. Os modelos de componentes utilizáveis pelo usuário (no estilo *drag and drop*) são `Tag`, `Toolbar`, `Menu`, `MenuItem`, `Page`, `SFrontE`, `SBackE`, `CssIcon`, `Panel`, `Window`, `Label`, `Textfield`, `Numberfield`, `Datefield`, `Textareafield`, `Combobox`, `Checkboxfield`, `Radiogroup`, `Button`, `Gridpanel`, `Column` e `Actioncolumn`. Estes modelos e suas propriedades estão especificados na seção 3.2.3.

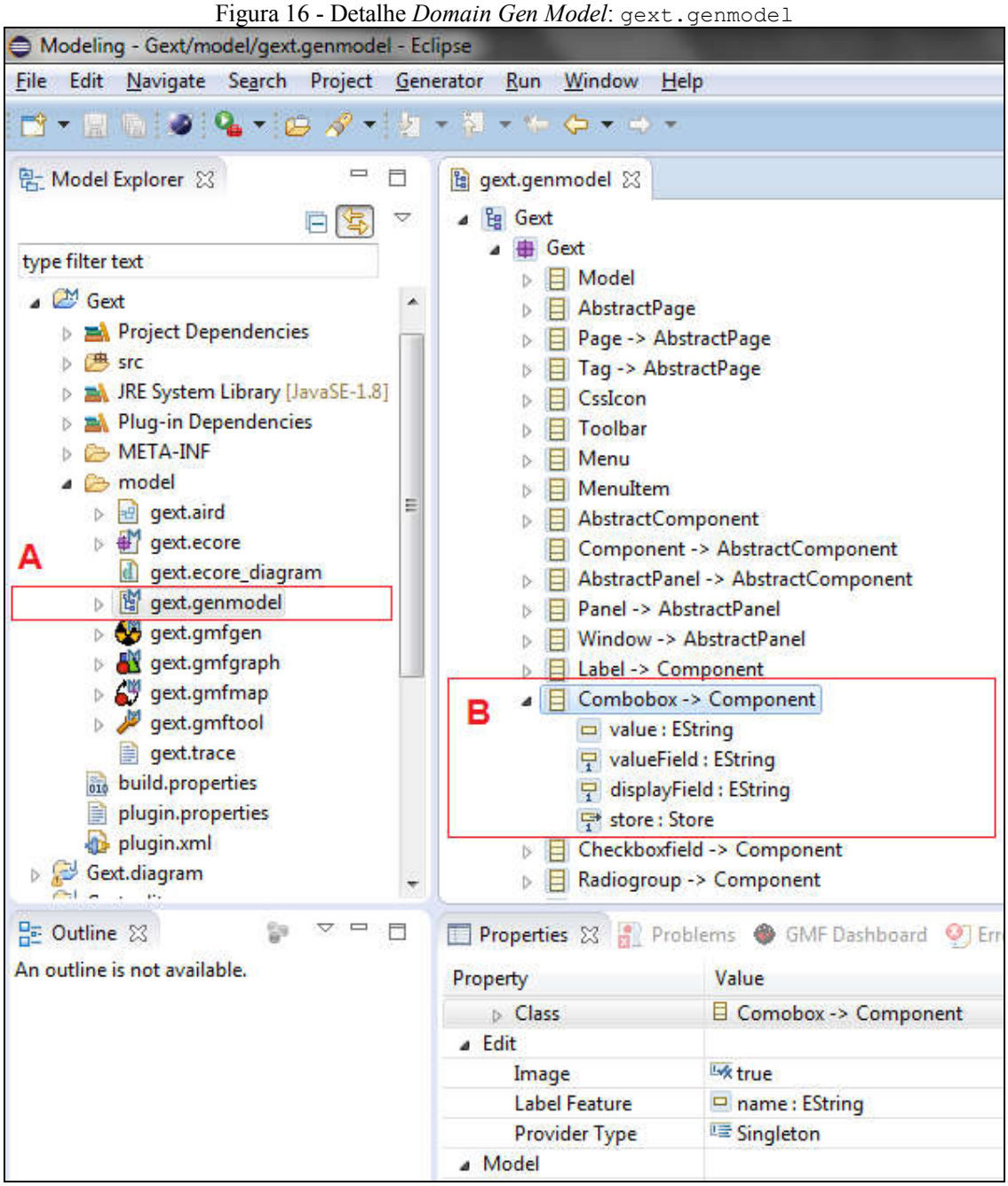
A Figura 15 mostra parte do conteúdo do *Domain Model*, que é o arquivo `gext.ecore` destacado na área "A". Esse arquivo contém todos os elementos citados anteriormente, porém alguns foram omitidos por motivos de simplificação da figura. Por exemplo, o componente `Combobox` e seus atributos são destacados na área "B". Os atributos especificados são os dados utilizados pelo gerador para criar o código fonte dos componentes nas páginas *web*.

Figura 15 - Detalhe *Domain Model*: `gext.ecore`



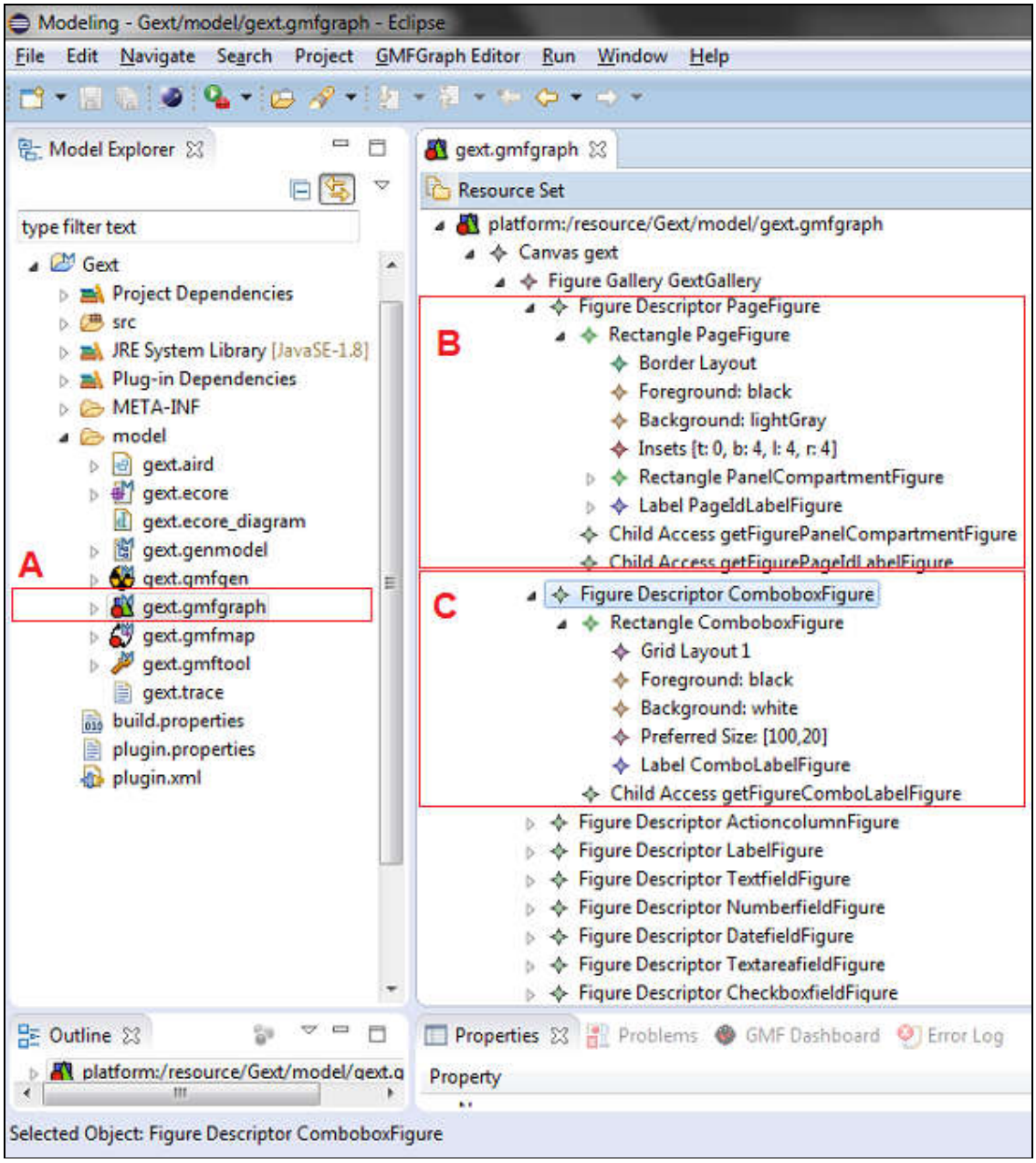
Conforme a Figura 16, observa-se que os mesmos componentes descritos no `gext.ecore` estão presentes no arquivo `gext.genmodel`, destacado na área "A". Este arquivo é o *Domain Gen Model*, responsável por gerar o código fonte dos *plugins* do modelo, do painel e do editor gráfico. Novamente o componente `Combobox` e suas propriedades são destacados na área "B". Outras opções customizáveis como por exemplo, a opção de um campo multi linha para a edição do usuário (`Property Multi-line`), ou inclusão de um componente contido em outro componente (`Create Children`) são definidas nele.

Figura 16 - Detalhe *Domain Gen Model*: `gext.genmodel`



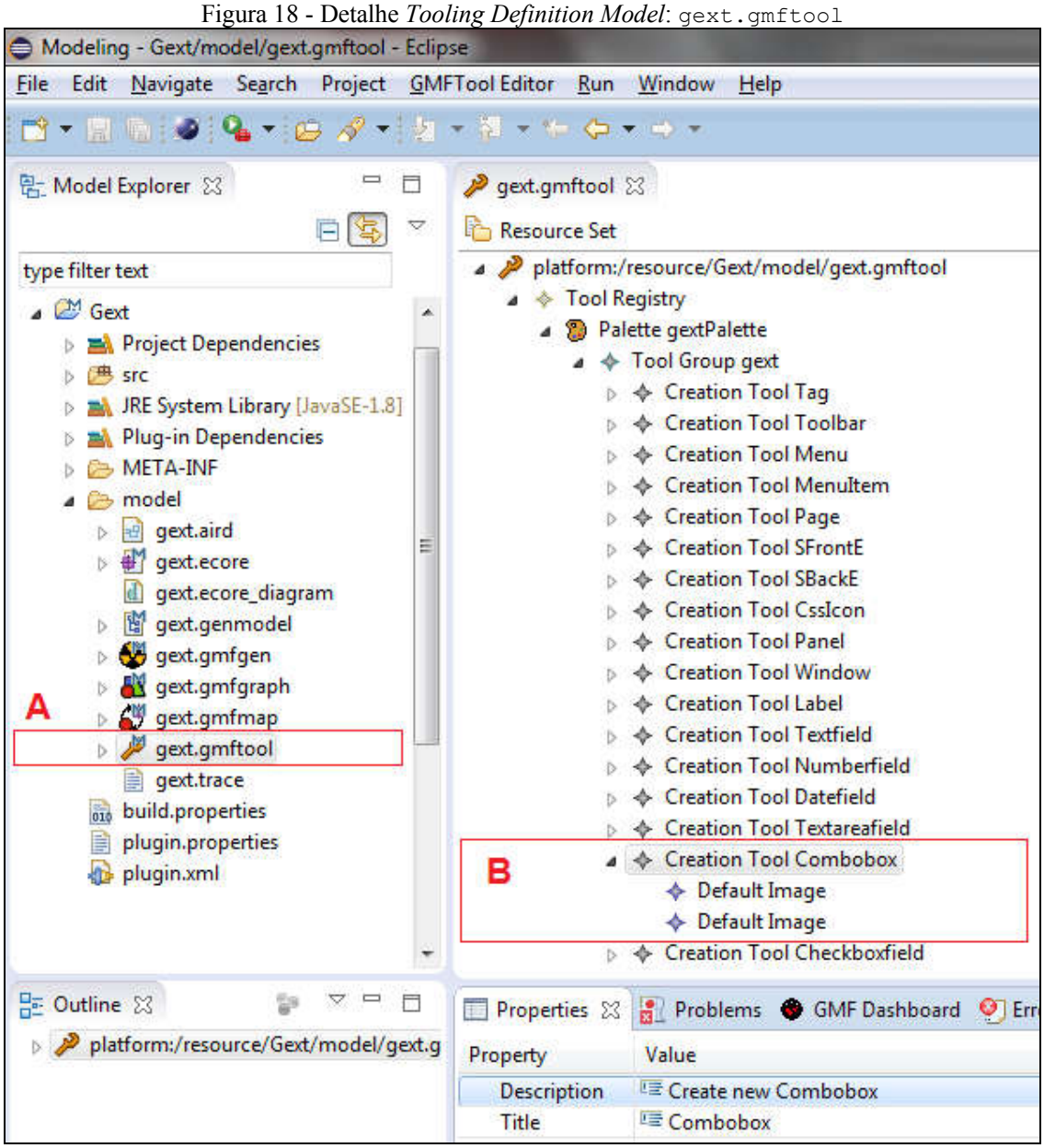
Na Figura 17 observa-se parte do conteúdo do *Graphical Definition Model*, (arquivo `gext.gmfgraph`) destacado na área "A". Esse arquivo define o formato do componente no editor gráfico. Em especial `Page`, que é uma página *web* destacada na área "B". Novamente o componente `Combobox` é visualizado na área destacada "C", definido como um retângulo de fundo branco, borda preta e com um rótulo dentro dele. Esse rótulo será posteriormente o atributo `id` definido no *Gen Model*, por meio do componente abstrato `AbstractComponent`.

Figura 17 - Detalhe *Graphical Definition Model*: `gext.gmfgraph`



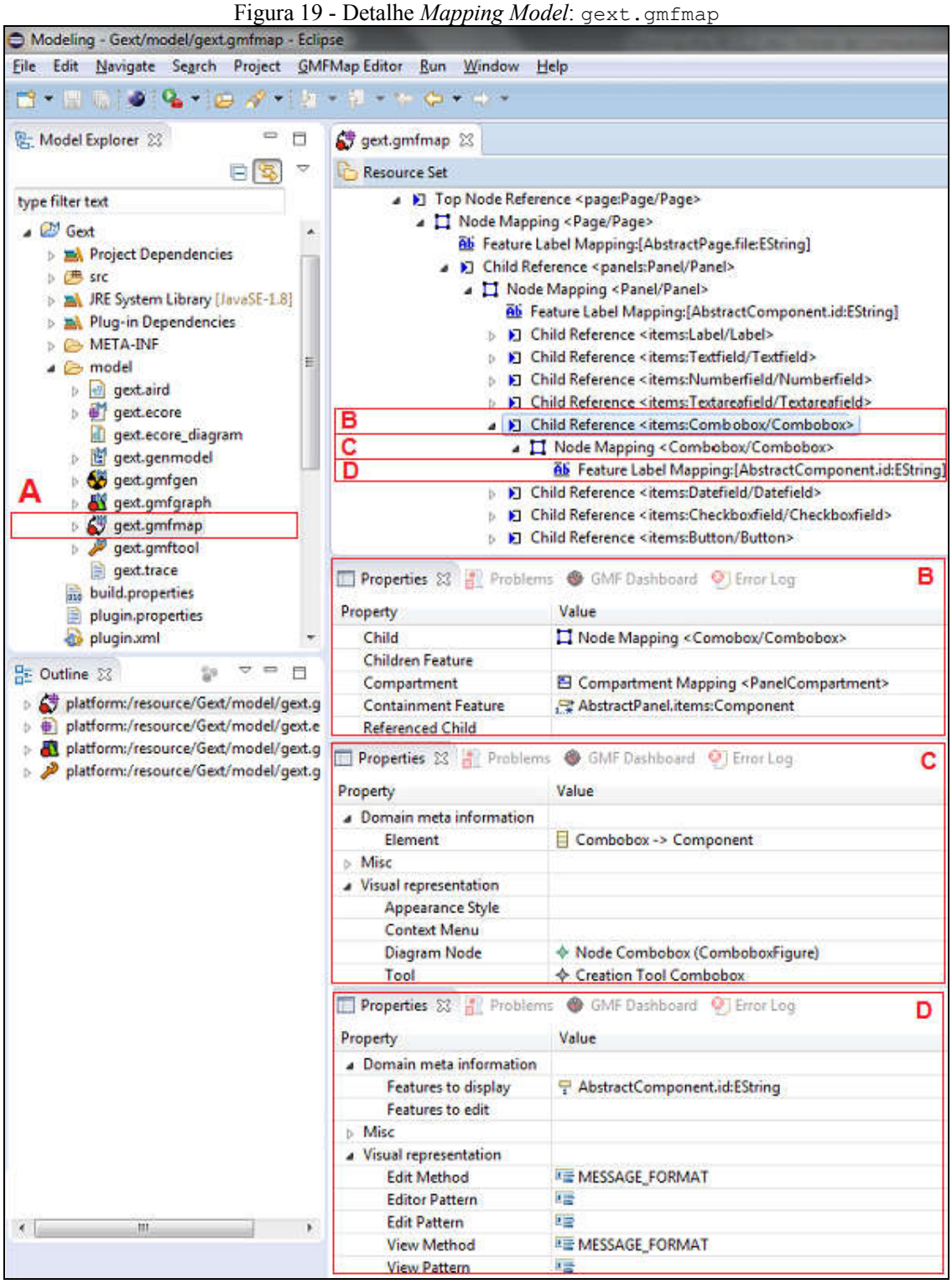
Conforme Figura 18, observa-se parte do conteúdo do *Tooling Definition Model*, que é o arquivo `gext.gmftool` destacado na área "A". Ele define a paleta de componentes para interação com o usuário, assim como a imagem, o título e descrição para apresentação ao usuário. Novamente o componente `Combobox` na área destacada "B" é exibido com detalhe para as imagens padrões que são fornecidas pelo Eclipse.

Figura 18 - Detalhe *Tooling Definition Model*: `gext.gmftool`



Conforme Figura 19, observa-se parte do conteúdo do *Mapping Model*, que é o arquivo `gext.gmfmap` destacado na área "A". Ele realiza a vinculação entre os elementos do *Domain Model* com os elementos do *Graphical Definition Model* e do *Tooling Definition Model*. Novamente o componente `Combobox` nas áreas destacadas "B", "C" e "D" é exibido com seu correspondente "B", "C" e "D" da aba *Properties* com detalhe para a vinculação entre os modelos *Domain Model*, *Graphical Definition Model* e *Tooling Definition Model*.

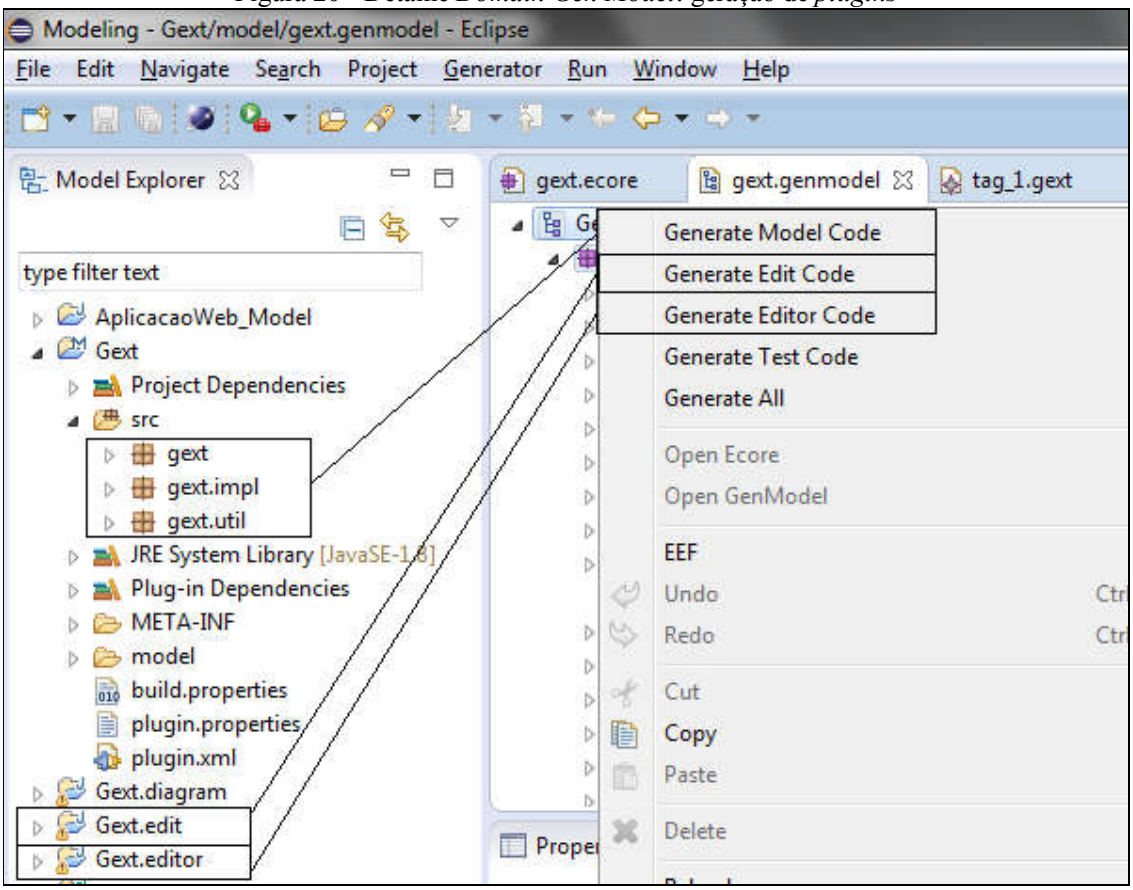
Figura 19 - Detalhe *Mapping Model*: `gext.gmfmap`



Conforme a Figura 20, a partir do *Domain Gen Model* foram criados os projetos para os *plugins* do modelo, do painel de propriedades e do editor gráfico. Tais projetos compõem apenas o editor gráfico, ou seja, a interface de modelagem com o usuário, portanto, não são responsáveis pela geração de código. Após a geração dos pacotes gext, gext.impl e gext.util

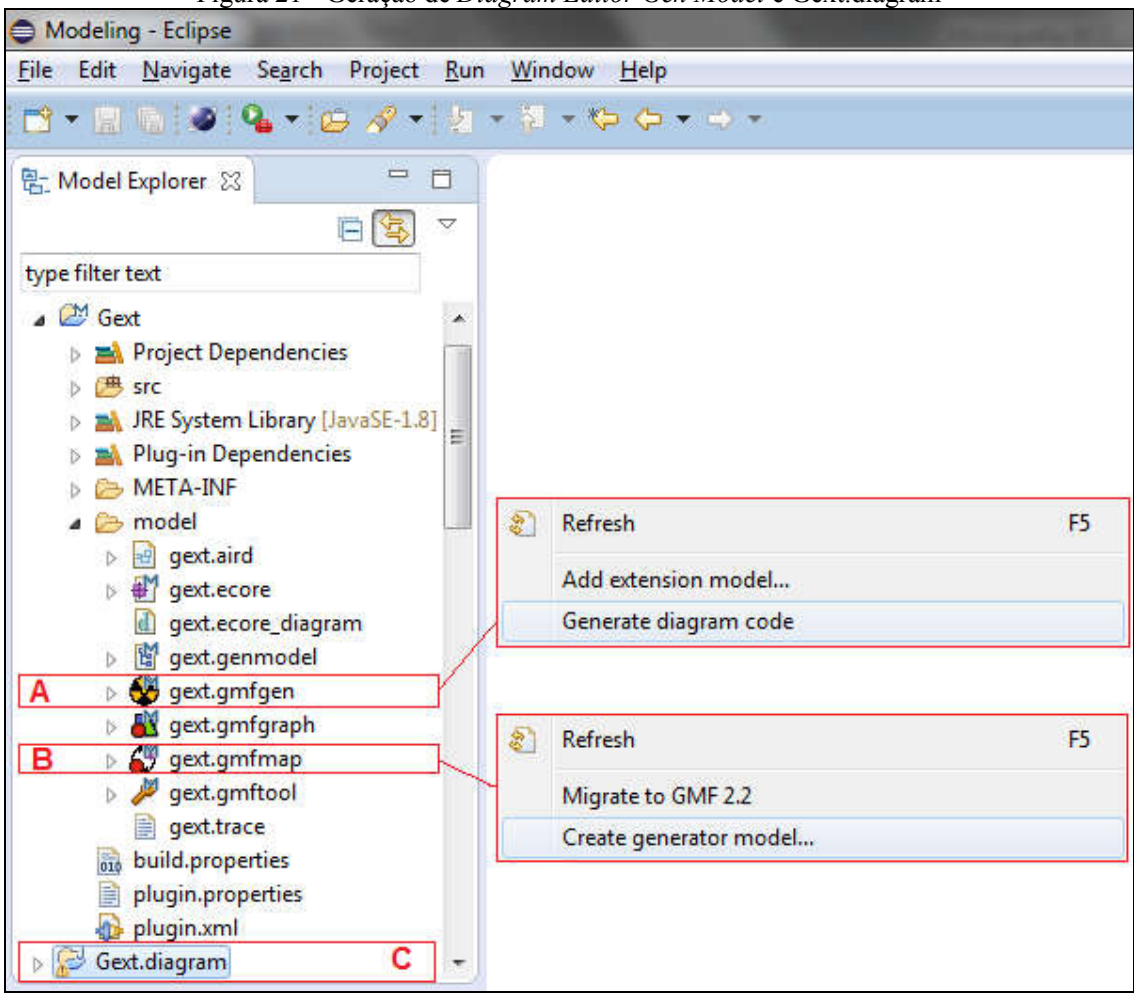
no projeto Gext, e dos projetos Gext.edit e Gext.editor, esses projetos foram exportados como *plugins* no Eclipse.

Figura 20 - Detalhe *Domain Gen Model*: geração de *plugins*



Conforme Figura 21, o *Diagram Editor Gen Model* é o arquivo `gext.gmfgen`, ele é gerado a partir da opção de *mouse* botão direito "*Create generator model...*" sob o arquivo `gext.gmfmap`, conforme área destacada "B". Após sua geração, por meio da opção de *mouse* botão direito "*Generate Diagram Code*" sob o arquivo `gext.gmfgen` destacado na área "A" é criado o projeto Gext.diagram destacado na área "C". Esse é o projeto final que une todos os projetos anteriores responsáveis por construir o editor gráfico do Gext JS. Este projeto foi exportado como *plugin* no Eclipse.

Figura 21 - Geração de *Diagram Editor Gen Model* e Gext.diagram



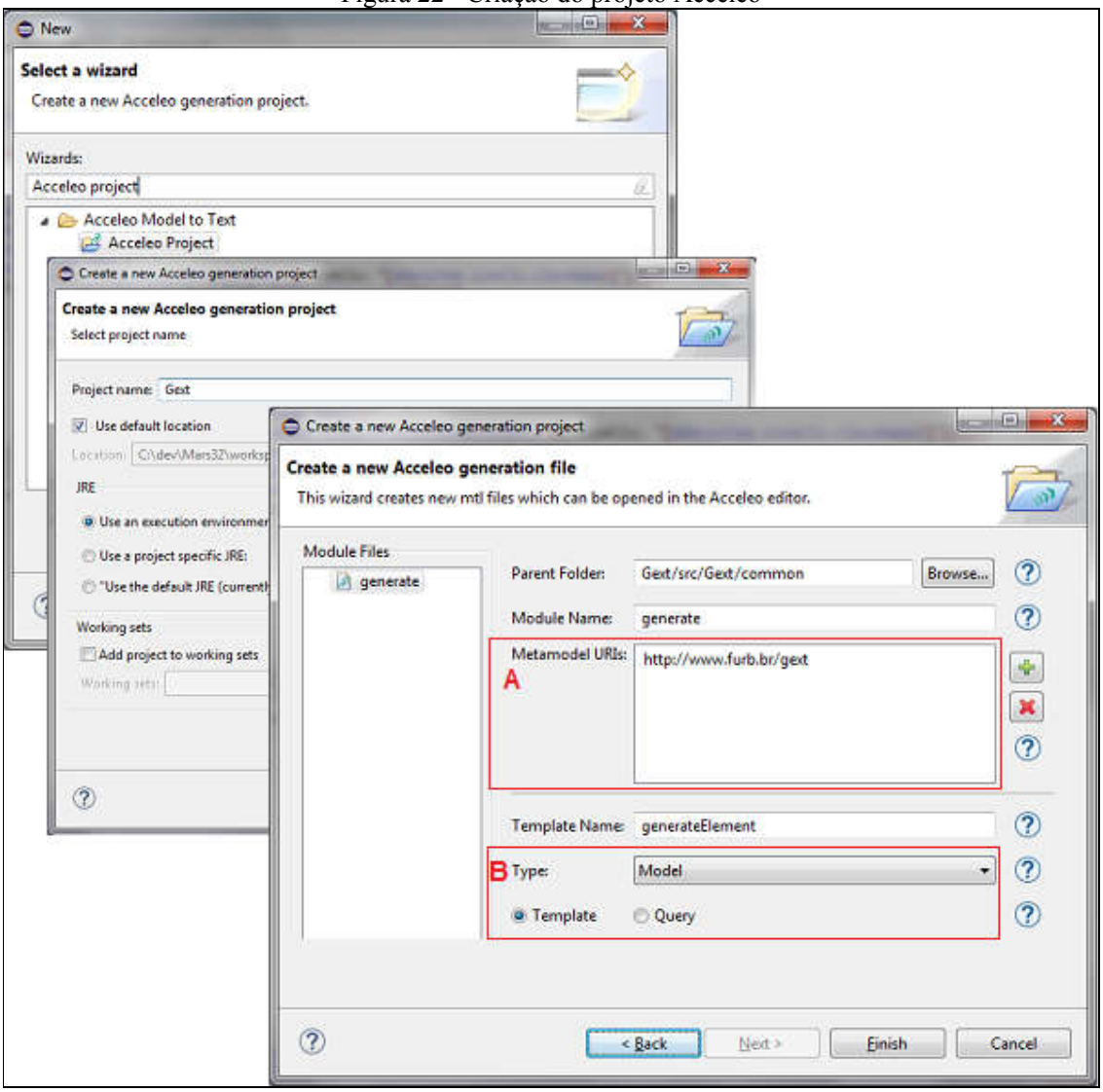
3.3.1.2 Construção do gerador de código do Gext JS

A ferramenta Acceleo é um *plugin* da IDE Eclipse utilizado para construir geradores de código do tipo *Model to Text* (M2T). Essa ferramenta oferece vantagens como capacidade de personalização e interoperabilidade, pois a construção de um gerador de código a partir do Acceleo é feita com a implementação de *templates* (THE ECLIPSE FOUNDATION, 2015a). É possível criar *templates* para arquivos de diversas linguagens, como Java, JavaScript, JSF, PHP, entre outras. A combinação desses *templates* com modelos criados com uma DSL construída com o uso do EMF e do GMF origina arquivos dessas linguagens (THE ECLIPSE FOUNDATION, 2015a).

Para a implementação do gerador de código do Gext JS, foi necessário criar um Projeto do tipo *Acceleo Project* chamado Gext.gen, por meio da opção de *mouse* botão direito "*New* > *Other* > *Acceleo Project*", conforme ilustra a Figura 22. Em destaque para a área "A", a *Uniform Resource Identifier* (URI) da DSL do Gext JS é indicada para os *templates*

reconhecerem os componentes criados na etapa de definição dos modelos, abordada na seção 3.3.1.1. Em destaque na área "B" está o modelo `Model`, que pertence à DSL do Gext JS. Esse elemento é o ponto inicial na varredura de componentes e é vinculado implicitamente com o *template* padrão `Main` do Acceleo. Também foi criado o projeto Gext.gen.ui por meio da opção de *mouse* botão direito "Acceleo > *Create* Acceleo UI *Launcher Project*, sob o projeto Gext.gen.

Figura 22 - Criação do projeto Acceleo

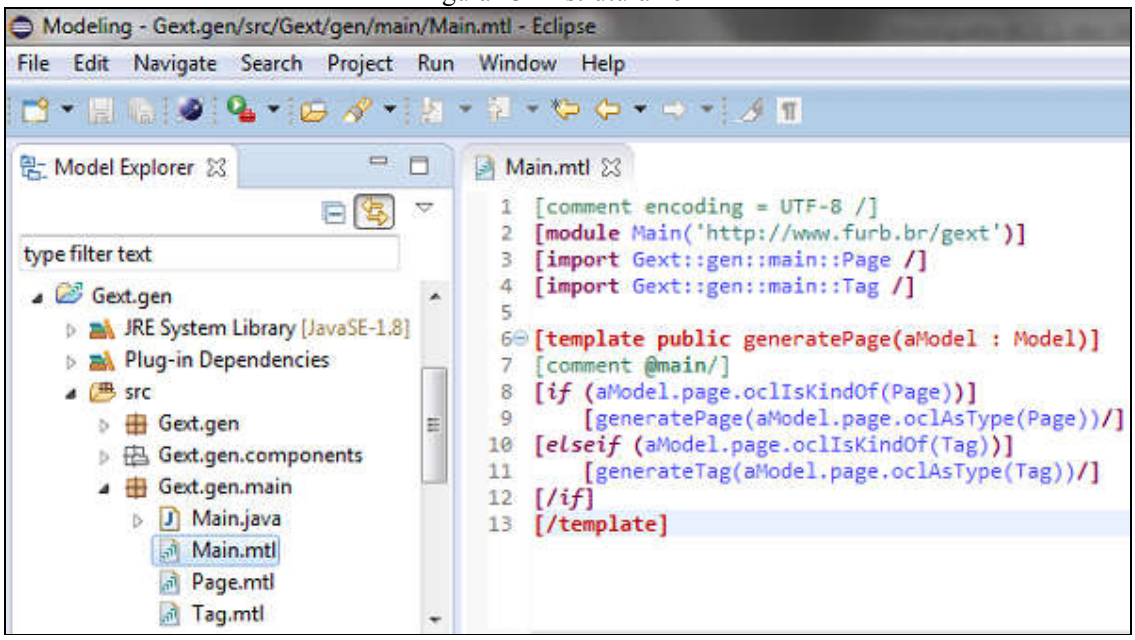


Após a etapa de criação de projetos, definiu-se os *templates* de geração de código. Para cada modelo criado no EMF e GMF, seu *template* correspondente também foi criado no Acceleo. A organização dos *templates* foi dividida em uma pasta chamada *main*, que armazena os *templates* iniciais de `Main` (*template* inicial do Acceleo), `Page` e `Tag`. Assim como uma pasta chamada *components* que armazena todos os demais *templates* correspondes

aos demais modelos do EMF e GMF. A criação dos *templates* foi feita pela opção de *mouse* botão direito "*New* > *Other...* > *Acceleo Module File*" sob a pasta desejada (*components* e/ou *main*). Cada *template* possui um componente da DSL ao qual está vinculado.

Conforme Figura 23, observam-se nas linhas 3 e 4 os *imports* dos modelos `Page` e `Tag`. A linha 6 representa a declaração do *template* que recebe como parâmetro um `Model`. Como `Model` é um modelo abstrato, verifica-se nas linhas 8 e 10 o tipo concreto, no caso, ou será `Page` ou será `Tag`. Para fins didáticos, é apresentada a sequência de geração de código de uma `Tag` com detalhe para os principais trechos de código, até o último componente gerado. Alguns trechos de código foram omitidos, pois seria inviável detalhar minuciosamente cada linha gerada.

Figura 23 - Estrutura `Main`



Na Figura 24 é apresentada a estrutura do *template* de `Tag`, a área destacada foi omitida para facilitar a visualização do *template* até o final. `Tag` faz uso da funcionalidade JSP (Java *Server Pages)* chamada *tag fragments*. Sendo assim, `Tag` é um *tag file*. Isso pode ser observado nas linhas 9, 10 e 93, nas quais são criados fragmentos para serem editados por outras páginas que a utilizam.

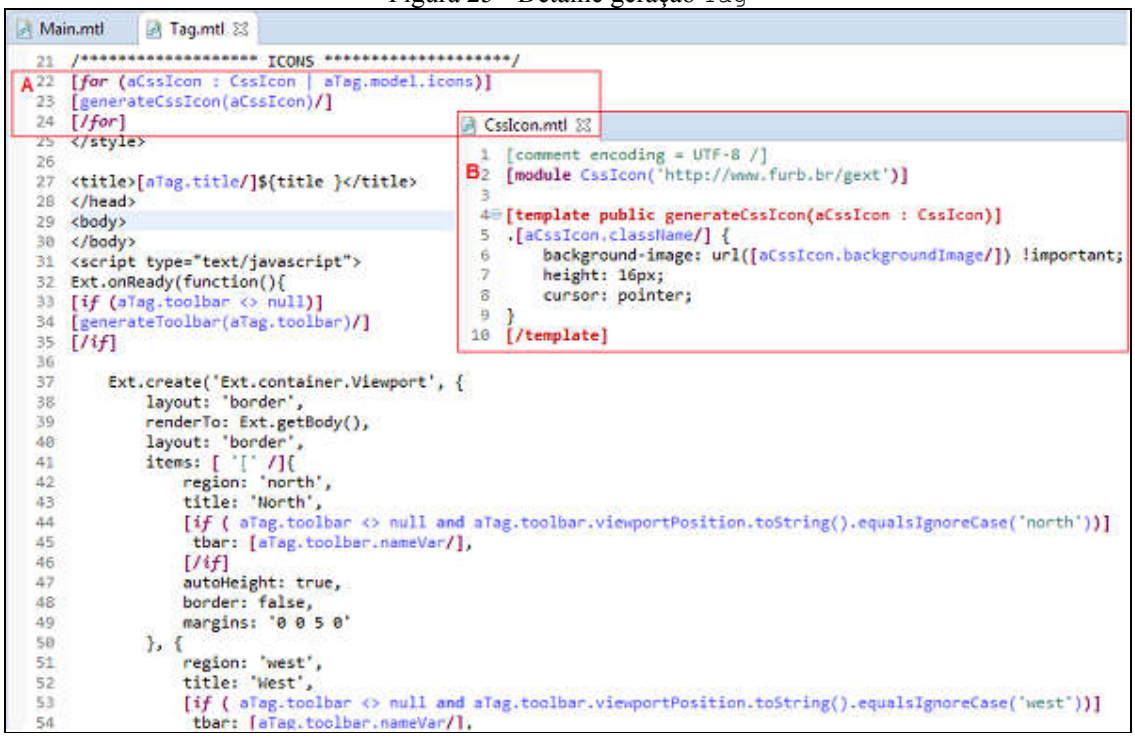
Figura 24 - Estrutura `Tag`

Tag.mtl

```
1  [comment encoding = UTF-8 /]
2  [module Tag('http://www.furb.br/gext')]
3  [import Gext::gen::components::Toolbar /]
4  [import Gext::gen::components::CssIcon /]
5
6  [template public generateTag(aTag : Tag)]
7  [file (aTag.file, false, 'UTF-8')]
8  <%@ tag language="java" pageEncoding="ISO-8859-1"%>
9  <%@ attribute name="title" required="true"  rtexprvalue="true"%>
10 <%@ attribute name="content" fragment="true"%>
11
12 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
13 <html>
14 <head>
15 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
16
17 <link rel="stylesheet" type="text/css" href="[aTag.hrefCssExtJS/]"/>
18 <script type="text/javascript" src="[aTag.hrefExtJS/]"></script>
19 <style type="text/css">
20
21 /****************** ICONS ******************/
22 [for (aCssIcon : CssIcon | aTag.model.icons)]
23 [generateCssIcon(aCssIcon)/]
24 [/for]
25 </style>
26
27 <title>[aTag.title/]${title }</title>
28 </head>
29 <body>
30 </body>
31 <script type="text/javascript">
32 Ext.onReady(function(){
    (...)
90 });
91 </script>
92
93 <jsp:invoke fragment="content"></jsp:invoke>
94
95 </html>
96
97
98 [/file]
99 [/template]
```

A Figura 25 exibe parte do *template* de geração da `Tag`. Observa-se, na área destacada "A", a iteração para todos os componentes `CssIcon` vinculados à `Tag`. Em detalhe destacado na área "B", encontra-se o *template* de `CssIcon` e sua geração conforme suas configurações. Na linha 33, é verificado se existe um `Toolbar`. Caso haja, o *template* para a geração do código correspondente é invocado. A partir da linha 37, é definido o *layout* padrão da `Tag` como `BorderLayout` e as vinculações necessárias com o `Toolbar` caso exista. A linha 44 verifica se o `Toolbar` deve ser adicionado na posição `north` do `BorderLayout`. Este mesmo tratamento pode ser observado na linha 53 para a posição `west` e assim por diante.

Figura 25 - Detalhe geração `Tag`

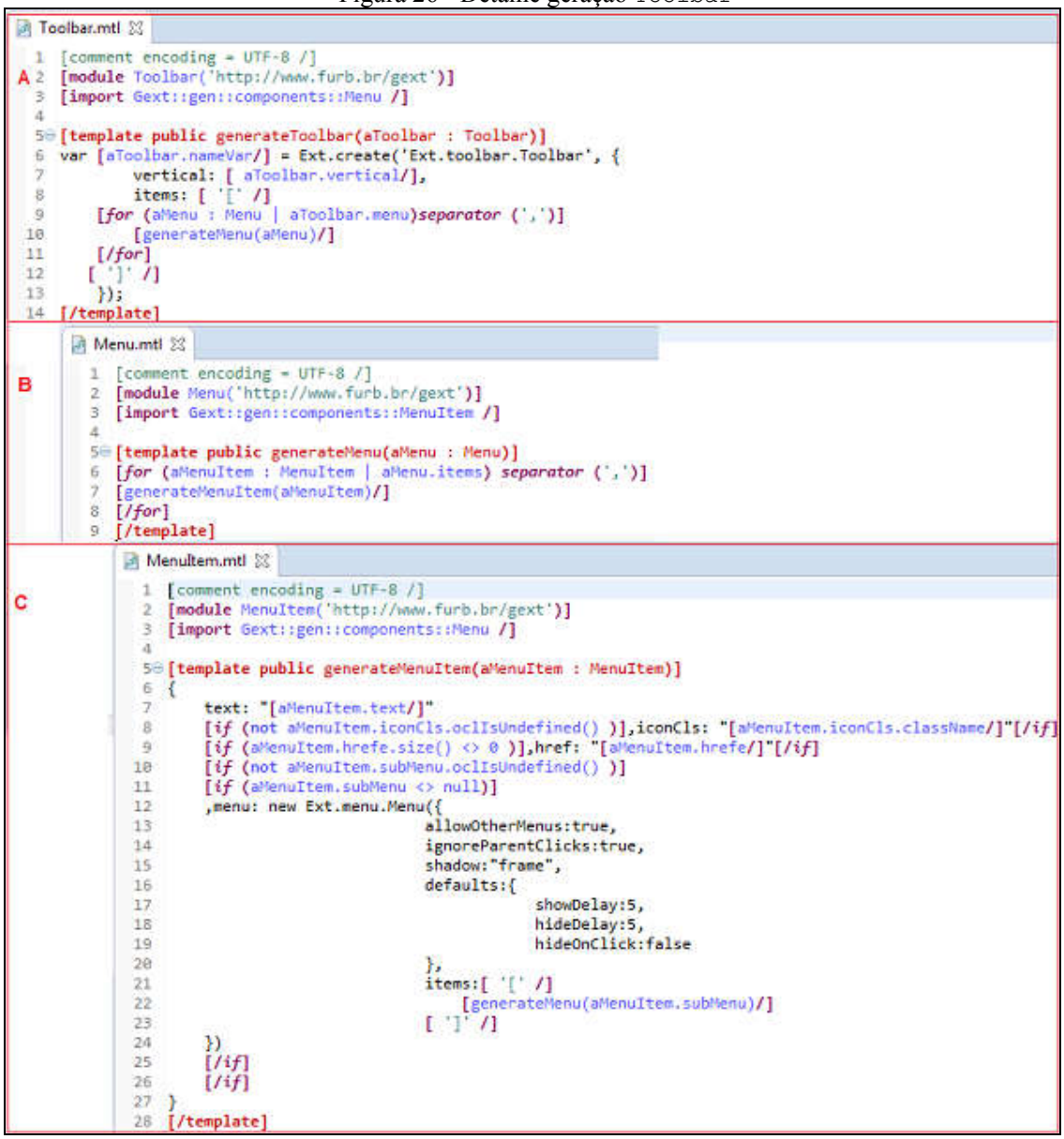



```
21 /******************** ICONS ********************/
22 [for (aCssIcon : CssIcon | aTag.model.icons)]
23 [generateCssIcon(aCssIcon)/]
24 [/for]
25 </style>
26
27 <title>[aTag.title/]${title }</title>
28 </head>
29 <body>
30 </body>
31 <script type="text/javascript">
32 Ext.onReady(function(){
33 [if (aTag.toolbar <> null)]
34 [generateToolbar(aTag.toolbar)/]
35 [/if]
36
37     Ext.create('Ext.container.Viewport', {
38         layout: 'border',
39         renderTo: Ext.getBody(),
40         layout: 'border',
41         items: [ '[' /]{
42             region: 'north',
43             title: 'North',
44             [if ( aTag.toolbar <> null and aTag.toolbar.viewportPosition.toString().equalsIgnoreCase('north'))]
45              tbar: [aTag.toolbar.nameVar/],
46             [/if]
47             autoHeight: true,
48             border: false,
49             margins: '0 0 5 0'
50         }, {
51             region: 'west',
52             title: 'West',
53             [if ( aTag.toolbar <> null and aTag.toolbar.viewportPosition.toString().equalsIgnoreCase('west'))]
54              tbar: [aTag.toolbar.nameVar/],
```

```
1 [comment encoding = UTF-8 /]
2 [module CssIcon('http://www.furb.br/gext')]
3
4 [template public generateCssIcon(aCssIcon : CssIcon)]
5 .[aCssIcon.className/] {
6     background-image: url([aCssIcon.backgroundImage/]) !important;
7     height: 16px;
8     cursor: pointer;
9 }
10 [/template]
```

Na área destacada "A" da Figura 26, observa-se a geração das propriedades do `Toolbar` e nas linhas 9, 10 e 11, a iteração para os componentes `Menu`, com detalhe para a linha 10, que faz a chamada para o *template* `Menu`. Na área destacada "B", há uma iteração para geração dos componentes `MenuItem`, com detalhe para a linha 7 que faz a chamada para o *template* `MenuItem`. Na área destacada "C", é mostrado o *template* de `MenuItem`. Um `MenuItem` pode conter outros componentes do tipo `Menu`. Assim, na linha 22 da área destacada "C", invoca-se novamente o *template* de `Menu`, para iniciar novamente interação para a geração `Menu` e `MenuItem`. Desta forma, é possível gerar um `ItemMenu` com subitens de seu mesmo tipo, ou seja, do tipo `ItemMenu`. Finaliza-se então a criação do *template* `Tag`.

Figura 26 - Detalhe geração `Toolbar`



Na Figura 27 é apresentada a estrutura do *template* de `Page`, as áreas destacadas foram omitidas para facilitar a visualização do *template* até o final. `Page` pode ou não fazer uso de uma `Tag`. Esse controle é feito na abertura do bloco, nas linhas 12 a 14, e no fechamento do bloco pelas linhas 61 a 64. Na área destacada "A", observa-se um laço de repetição para cada *template* do tipo `Panel`, `Window` e `Gridpanel`.

Figura 27 - Estrutura `Page`

Page.mtl

```
1  [comment encoding = UTF-8 /]
2  [module Page('http://www.furb.br/gext')]
3  [import Gext::gen::components::Panel /]
4  [import Gext::gen::components::Window /]
5  [import Gext::gen::components::CssIcon /]
6  [import Gext::gen::components::Gridpanel /]
7
8  [template public generatePage(aPage : Page)]
9  [file (aPage.file, false, 'UTF-8')]
10 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
11     pageEncoding="ISO-8859-1"%>
12 [if (aPage.tag.size() <> 0)]
13 <%@ taglib prefix="mt" tagdir="[aPage.tagdir/]"%>
14 [/if]
15 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
16 <html>
17 <head>
18 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
19
20 <link rel="stylesheet" type="text/css" href="[aPage.hrefCssExtJS/]"/>
21 <script type="text/javascript" src="[aPage.hrefExtJS/]"></script>
22 <style type="text/css">
   (...)
35     <mt:[aPage.tag/] title="[aPage.title /]">
36     <jsp:attribute name="content">
37         <script type="text/javascript" >
38                 Ext.onReady(function(){
   (...)
47 [for (aPanel : Panel | aPage.panels)]
48     [generatePanel(aPanel.oclAsType(Panel))/]
49 [/for]
50 [for (aWindow : Window | aPage.windows)]
51     [generateWindow(aWindow.oclAsType(Window))/]
52 [/for]
53 [for (aGridpanel : Gridpanel | aPage.grid)]
54     [generateGridpanel(aGridpanel.oclAsType(Gridpanel))/]
55 [/for]
56
57 [if (aPage.tag.size() <> 0)]
58 });
59 [/if]
60 </script>
61 [if (aPage.tag.size() <> 0)]
62     </jsp:attribute>
63 </mt:[aPage.tag/]>
64 [/if]
65
66 </html>
67
68 [/file]
69 [/template]
```

A

Na Figura 28 observa-se na área destacada "A" o laço de repetição em que são buscados os componentes vinculados à `Window`. Dentro do laço é verificado o tipo de componente e é invocado o seu respectivo *template* para a geração de código. O laço de repetição de `Panel` é similar ao laço de `Window`.

Figura 28 - Iteração de `Component`

Window.mtl

```
1  [comment encoding = UTF-8 /]
2  [module Window('http://www.furb.br/gext')]
3  [import Gext::gen::components::Button /]
   (...)
11 [import Gext::gen::components::Radiogroup /]
12
13
14 [template public generateWindow(aWindow : Window)]
15 [if (not aWindow.insideMethodName.oclIsUndefined())]
16     var [aWindow.insideMethodName/] = function(){
17 [/if]
18     Ext.create("Ext.window.Window", {
19     id: "[aWindow.id/]"
20     [if (aWindow.name.size() <> 0)],name: "[aWindow.name/]"[/if]
28     [if (not aWindow.formId.oclIsUndefined())]
29     ,items: [ '[' /]
30     {
31         xtype: "form"
32         [if (aWindow.formId.size() <> 0)],id: "[aWindow.formId/]"[/if]
33         [if (aWindow.formName.size() <> 0)],name: "[aWindow.formName/]"[/if]
   (...)
40         [if (aWindow.formUrl.size() <> 0)],url: [aWindow.formUrl/][/if]
41         ,items: [ '[' /]
42         [for (aComponent : Component | aWindow.items) separator (',')]
43             [if (aComponent.oclIsTypeOf(Label))]
44                 [generateLabel(aComponent.oclAsType(Label))/]
45             [elseif (aComponent.oclIsTypeOf(Combobox))]
46                 [generateComboBox(aComponent.oclAsType(Combobox))/]
47             [elseif (aComponent.oclIsTypeOf(Textfield))]
48                 [generateTextfield(aComponent.oclAsType(Textfield))/]
49             [elseif (aComponent.oclIsTypeOf(Numberfield))]
50                 [generateNumberfield(aComponent.oclAsType(Numberfield))/]
51             [elseif (aComponent.oclIsTypeOf(Datefield))]
52                 [generateDatefield(aComponent.oclAsType(Datefield))/]
53             [elseif (aComponent.oclIsTypeOf(Textareafield))]
54                 [generateTextareafield(aComponent.oclAsType(Textareafield))/]
55             [elseif (aComponent.oclIsTypeOf(Checkboxfield))]
56                 [generateCheckboxfield(aComponent.oclAsType(Checkboxfield))/]
57             [elseif (aComponent.oclIsTypeOf(Radiogroup))]
58                 [generateRadiogroup(aComponent.oclAsType(Radiogroup))/]
59             [elseif (aComponent.oclIsTypeOf(Button))]
60                 [generateButton(aComponent.oclAsType(Button))/]
61             [/if]
62         [/for]
63     [ ']'  /]/*end items-form*/
64     }
65     [ ']' /]
66     [/if]
67     ,buttons: [ '['  /]
68     [for (aButton : Button | aWindow.buttons) separator (',')]
69         [generateButton(aButton)/]
70     [/for]
71         [ ']'  /]
72 })[if (aWindow.isShow)].show()[/if];
73 [if (not aWindow.insideMethodName.oclIsUndefined())]
74     }
75 [/if]
76 [/template]
```

A

Um exemplo de invocação de um *template* do tipo `Component` pode ser visualizado na Figura 29. Observa-se, na área destacada "A", o *template* de `Combobox` e na linha 19 a invocação para o *template* `SFrontE`, exibido na área destacada "B". Os *templates* dos demais componentes seguem uma estrutura semelhante. Após a criação de todos os *templates*, os projetos Gext.gen e Gext.gen.ui foram exportados como *plugins* do Eclipse.

Figura 29 - Detalhe `Combobox` e `SFrontE`

ComboBox.mtl

A

```
1  [comment encoding = UTF-8 /]
2  [module ComboBox('http://www.furb.br/gext')]
3  [import Gext::gen::components::SFrontE /]
4  [import Gext::gen::components::SBackE /]
5
6  [template public generateComboBox(aCombobox: Combobox)]
7  {
8      xtype: "combo"
9      ,id: "[aCombobox.id/]"
10     [if (aCombobox.name.size() <> 0)],name: "[aCombobox.name/]"[/if]
11     [if (aCombobox.x.size() <> 0)],x: [aCombobox.x/][/if]
12     [if (aCombobox.y.size() <> 0)],y: [aCombobox.y/][/if]
13     [if (aCombobox.width.size() <> 0 )],width: [aCombobox.width/][/if]
14     [if (aCombobox.height.size() <> 0 )],height: [aCombobox.height/][/if]
15     [if (aCombobox.valueField.size() <> 0)],valueField: "[aCombobox.valueField/]"[/if]
16     [if (aCombobox.displayField.size() <> 0)],displayField:"[aCombobox.displayField/]"[/if]
17     [if (aCombobox.value.size() <> 0)],value: [aCombobox.value/][/if]
18     [if (aCombobox.store.oclIsTypeOf(SFrontE))]
19         ,store:[generateSFrontE(aCombobox.store.oclAsType(SFrontE))/]
20     [elseif (aCombobox.store.oclIsTypeOf(SBackE))]
21         ,store:[generateSBackE(aCombobox.store.oclAsType(SBackE))/]
22     [/if]
23 }
24 [/template]
```

SFrontE.mtl

B

```
1  [comment encoding = UTF-8 /]
2  [module SFrontE('http://www.furb.br/gext')]
3
4
5  [template public generateSFrontE(aSFrontE : SFrontE)]
6  new Ext.data.Store({
7                  storeId: "[aSFrontE.storeId/]"
8                  ,autoLoad: [aSFrontE.autoLoad/]
9                  [if (aSFrontE.pageSize.size() <> 0 )]
10                 ,pageSize: [aSFrontE.pageSize/]
11                 [/if]
12                 ,fields: [ '[' /]
13                     [aSFrontE.fields/]
14                 [ ']' /]
15                 ,data:[ '[' /]
16                     [aSFrontE.data/]
17                 [ ']' /]
18                 ,listeners: {
19                             beforesync: function( options, eOpts ){}
20                             [if (aSFrontE.listeners_beforeload.size() <> 0)]
21                             ,beforeload: function(s,o){
22                                 [aSFrontE.listeners_beforeload/]
23                             }
24                             [/if]
25                             [if (aSFrontE.listeners_load.size() <> 0)]
26                             ,load: function(s,r,o){
27                                 [aSFrontE.listeners_load/]
28                             }
29                             [/if]
30                 }
31 })
32 [/template]
```

3.3.2 Operacionalidade da ferramenta

Para utilizar o Gext JS, recomenda-se utilizar as ferramentas descritas na seção 3.3.1 e como complemento a estas ferramentas deve-se adicionar ao Eclipse os seguintes itens:

a) Ext JS: incluir o Ext JS versão 4.0.7 disponibilizado pela Sencha em seu site oficial e configurá-lo. Por exemplo, para esse trabalho foi criado um projeto *web* de teste chamado Site1. Na sua pasta WebContent foi criada o caminho de pasta "extjs/resources" e foram incluídos os arquivos `ext-all.js` e `ext-all.js` do Ext JS obtido;
b) Gext JS : incluir os *plugins* oriundos do Gext JS chamados `Gext.diagram_1.0.0.jar`, `Gext.gen_1.0.0.jar`, `Gext.gen.ui_1.0.0.jar`, `Gext.edit_1.0.0.jar`, `Gext.editor_1.0.0.jar`, `Gext_0.1.0.jar`. Estes *plugins* podem ser incluídos na pasta principal do Eclipse, por exemplo "C:\dev\Mars32\eclipse\plugins";
c) EMF/GMF: instalar os *plugins* do EMF/GMF;
d) Acceleo: instalar os *plugins* do Acceleo.

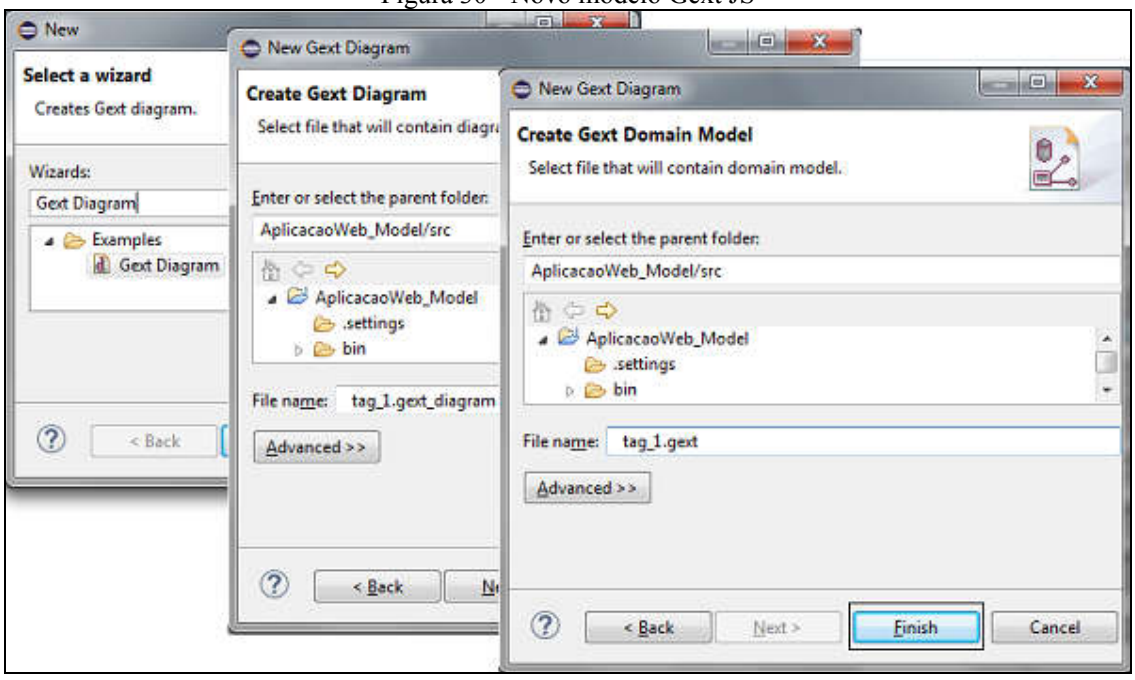
Para iniciar o desenvolvimento, é necessário já ter um certo conhecimento na ferramenta Ext JS, pois os componentes que o Gext JS fornece são semelhantes aos disponíveis no Ext JS. Como é o caso do componente `Store`, que possui resumidamente as propriedades: `storeId`, `autoLoad`, `pageSize` e `fields`. Se comparado ao componente nativo do Ext JS chamado `Ext.data.Store`, essas mesmas propriedades são encontradas na sua documentação. Após a etapa de configuração, o usuário poderá modelar a página *web* visualmente, por meio de uma paleta de componentes fornecida no Eclipse. Essa modelagem pode ser feita por meio de uma notação tabular/árvore, em que o usuário utiliza uma árvore de componentes que representa os elementos da página *web*. Assim como a modelagem gráfica, em que pode-se acompanhar em tempo de modelagem, como a aparência da página será gerada.

Ao modelar uma página no editor gráfico do Gext JS, são criados dois arquivos. O primeiro possui extensão ".gext" e contém somente os dados necessários para a geração de código. Quando aberto no editor, esse arquivo apresenta uma notação em árvore para ilustrar o conteúdo da página *web* modelada. O segundo arquivo, com extensão ".gext_diagram", apresenta a notação de modelagem com uma paleta de componentes no estilo *drag and drop*. Porém, esse segundo arquivo é apenas uma representação gráfica do arquivo ".gext" e não é utilizado na geração de código.

Para melhor compreensão, serão descritos os passos a serem seguidos para iniciar o desenvolvimento de um novo modelo. Criou-se um projeto chamado AplicacaoWeb_Model dentro do Eclipse *Modeling Tools*, para modelar a página *web*. Após a modelagem e geração de código, os arquivos gerados são copiados e inseridos no projeto *web* final, neste caso, o Site1. Inicialmente será descrita a modelagem de uma `Tag`, o nome deste arquivo será `tag_1.gext` e `tag_1.gext_diagram` para fins didáticos. Em seguida, será descrita a modelagem de uma `Page` que faz uso desta `Tag`. Esta `Page` conterá um formulário com alguns componentes disponíveis na ferramenta. A nomenclatura desta `Page` será `page_1.gext` e `page_1.gext_diagram`. Sendo assim, esta seção trata-se de um manual do Gext JS.

Na Figura 30 é mostrado o procedimento para abrir o editor de modelagem de páginas do Gext JS. Para isso, utiliza-se a opção de *mouse* botão direito sob a pasta src (por exemplo): "*New* > *Other...* > GextDiagram". Deve-se definir os nomes dos arquivos ".gext" e ".gext_diagram". Pressiona-se o botão *Next*, em seguida o botão *Finish*. Apesar de não ser obrigatório, recomenda-se que esses dois arquivos sejam criados com o mesmo nome, diferenciando-se apenas na extensão.

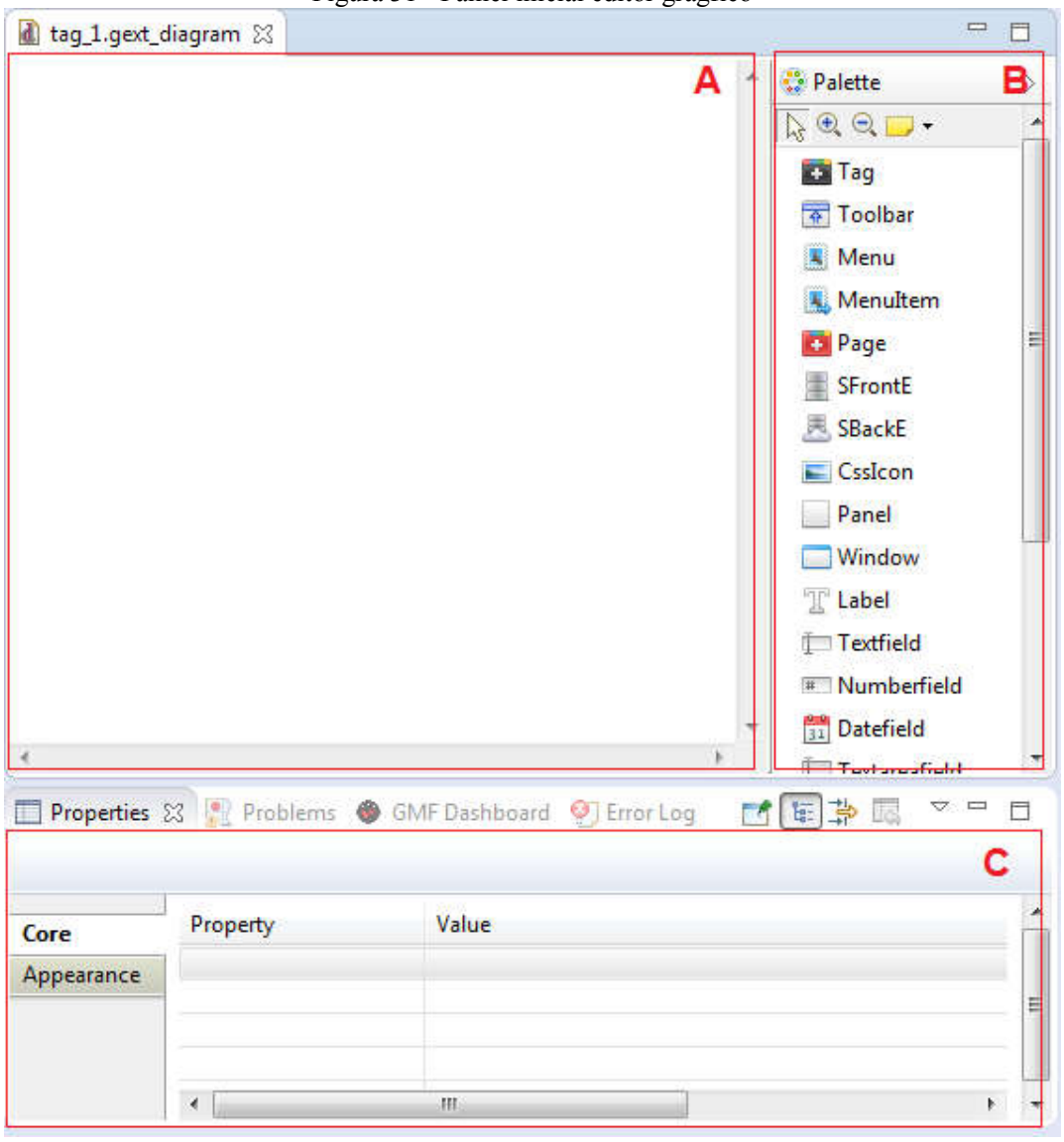
Figura 30 - Novo modelo Gext JS



Conforme a Figura 31, a área destacada "A" refere-se ao painel onde são dispostos os componentes. Toda a área em branco "A" é o componente principal `Model`, ele já é incluído por padrão ao iniciar um modelo, desta forma não é possível ser incluído pelo usuário. A área destacada "B" refere-se à paleta de componentes lateral direita *Palette*. Nela encontram-se todos os componentes disponíveis para modelagem. A área destacada "C" refere-se ao menu

de propriedades *Properties* da aba padrão do Eclipse. Edita-se o valor de uma propriedade de um determinado componente por esta aba. Sendo assim, para utilizar um componente, basta pressionar com o *mouse* botão esquerdo sob ele, na área "B", e arrastá-lo para o quadro branco "A". As propriedades do componente podem ser editadas na aba inferior *Properties*, destacada na área "C".

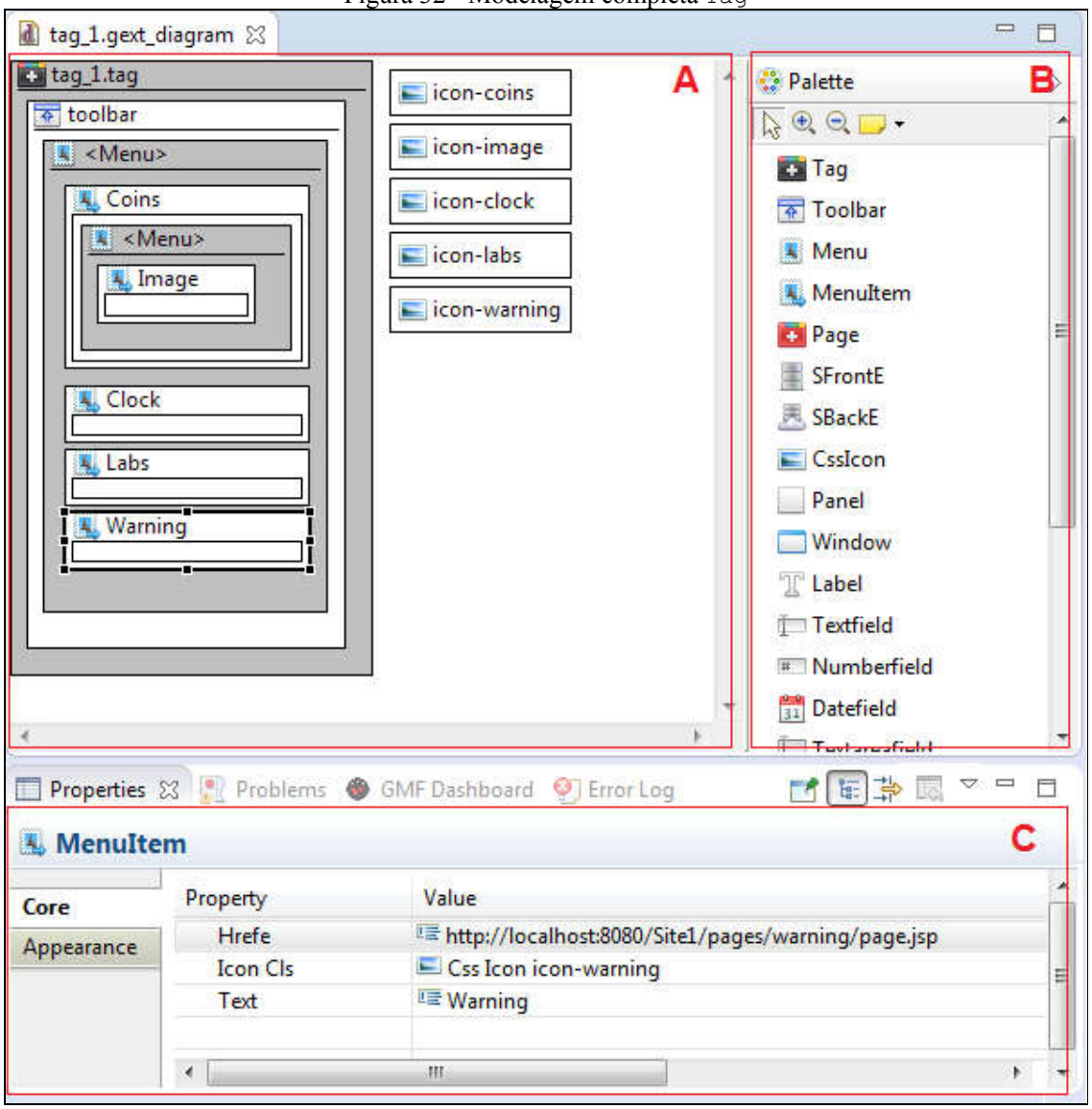
Figura 31 - Painel inicial editor grágfico



Para exemplo do uso da `Tag`, é desenvolvida uma página *web* que contém um *layout* padrão do tipo borda (norte, sul, leste, oeste, centro), uma barra de navegação com ícones de menu e demais componentes secundários. Na Figura 32 é mostrado este exemplo modelado. Foram utilizados os componentes `Model` (padrão), `Tag`, `Toolbar`, `Menu`, `MenuItem` e `CssIcon`. Esse modelo, representa uma página padrão com um menu de navegação. A seguir

descrevem-se os componentes utilizados do tipo `MenuItem`. Entre parênteses estão os sub menus, contidos dentro de `MenuItem`, são eles `Coins` (`Image`), `Image`, `Clock`, `Labs` e `Warning`.

Figura 32 - Modelagem completa `Tag`



Ainda na Figura 32, observa-se o menu de propriedades *Properties* que faz a vinculação entre os componentes e suas definições. O item de menu (`MenuItem`, `Warning`) está configurado da seguinte forma:

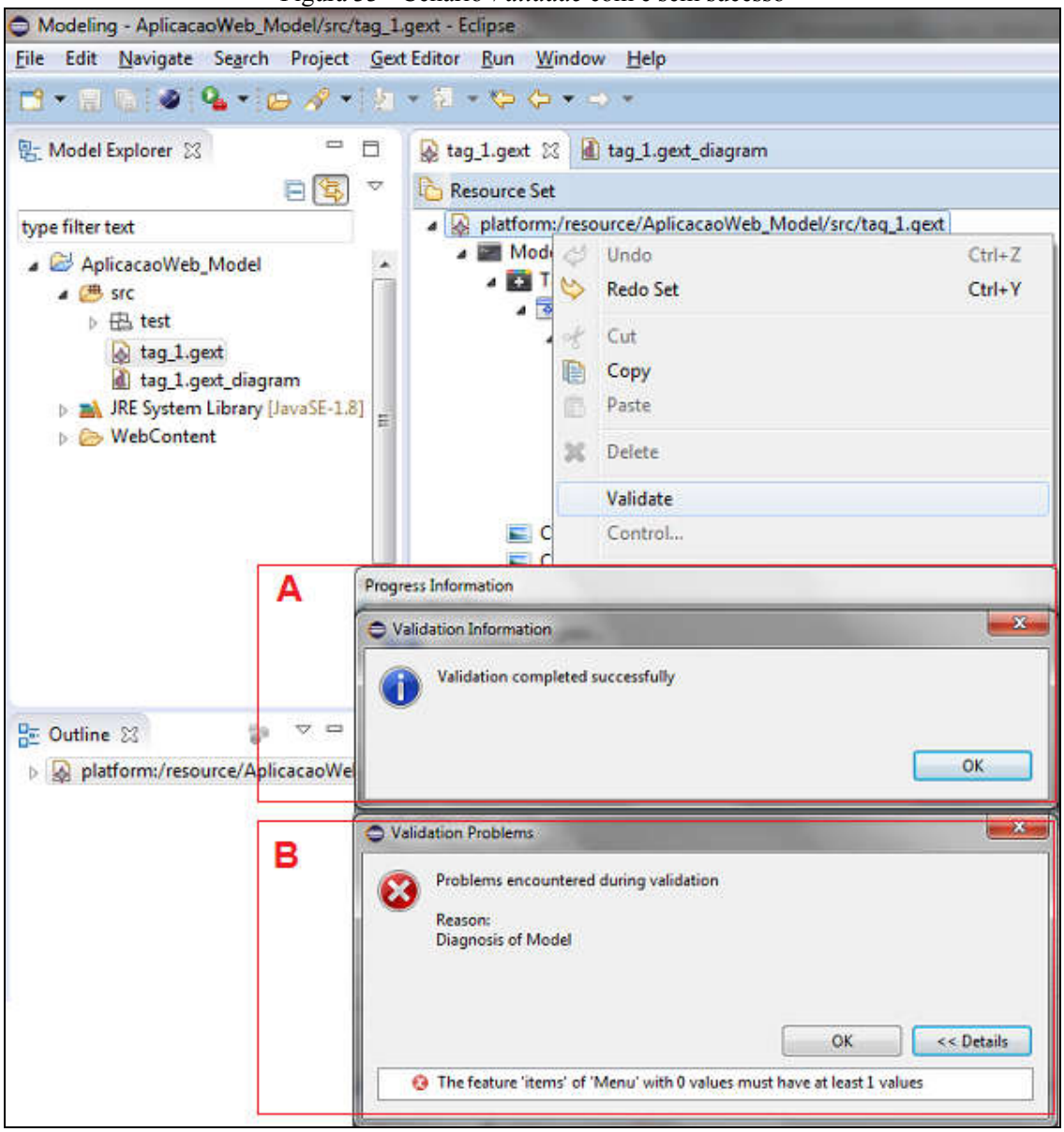
a) por meio da propriedade `Hrefe`: possui um redirecionamento para a página http://localhost:8080/Site1/pages/warning/page.jsp;

b) por meio da propriedade `IconCls`: possui vinculação com o ícone *icon-warning*;

c) por meio da propriedade `Text`: possui seu texto de exibição definido como *Warning*.

Após a modelagem da `Tag`, conforme a Figura 33, recomenda-se executar o comando *Validate* no arquivo tag_1.gext. Para isso, utiliza-se o *mouse* botão direito sob o arquivo `tag_1.gext` e seleciona-se a opção *Validate*. Desta forma, caso todos os componentes estejam configurados corretamente, uma mensagem de sucesso será exibida, conforme a área destacada "A". Porém, caso haja algum componente configurado incorretamente, uma mensagem de erro será informada, conforme área destacada "B". Nessa mensagem o erro é informado em inglês, pois trata-se de uma mensagem padrão do Eclipse. Para esse exemplo, a mensagem informa que o componente `Menu` deve conter ao menos um `MenuItem` obrigatoriamente.

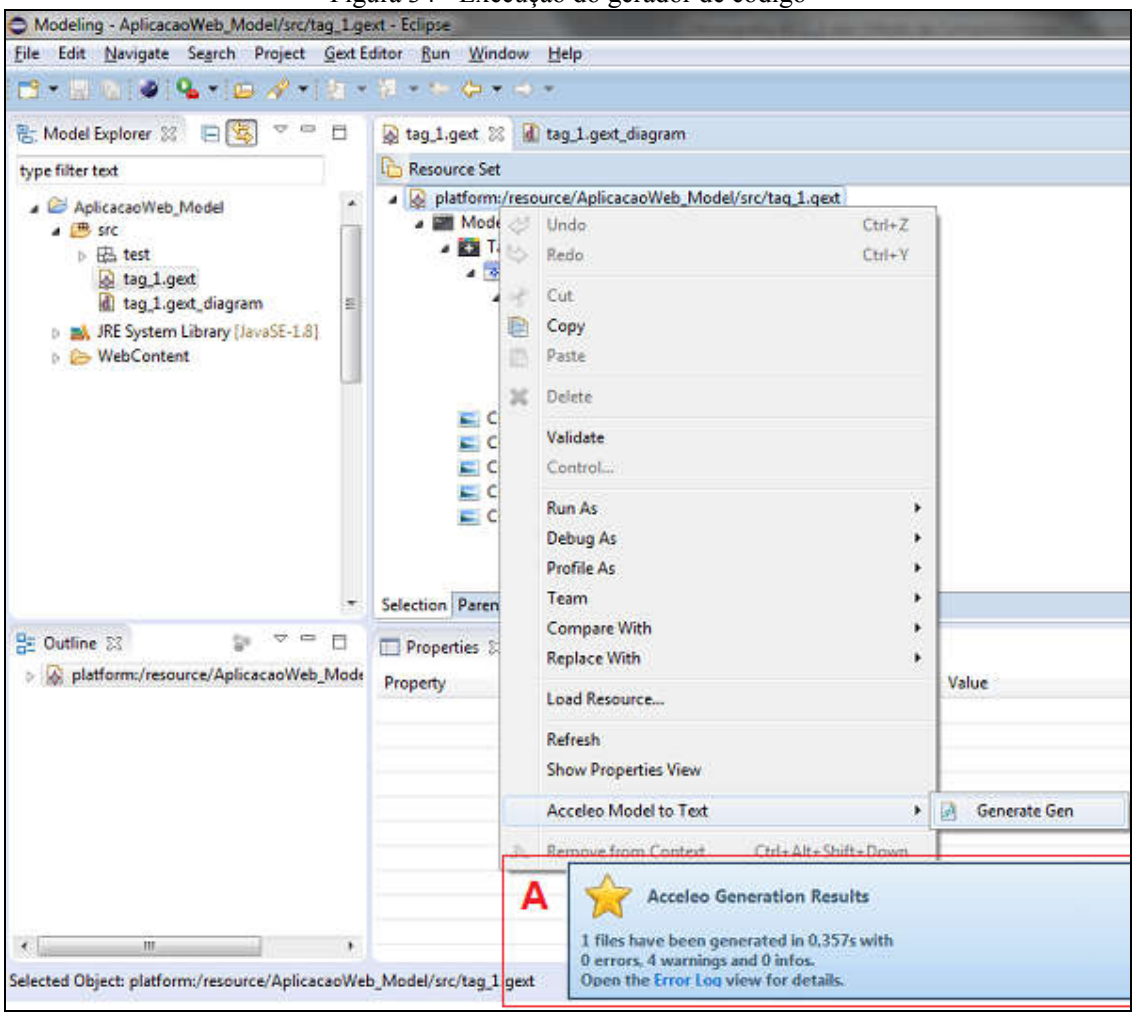
Figura 33 - Cenário *Validade* com e sem sucesso



Mesmo que o comando *Validate* retorne erro, é possível executar o gerador de código. Contudo, possivelmente as propriedades ou componentes que apresentaram problemas serão inconsistentes, o que resultará em defeito no código gerado. Nesse caso, o usuário pode editar o código gerado, caso queira utilizá-lo corretamente, porém a opção mais recomendada é corrigir os problemas de validação antes de gerar o código.

Para executar o gerador de código, conforme Figura 34, utiliza-se o *mouse* botão direito sob o arquivo `tag_1.gext` e seleciona-se a opção "Acceleo *Model to Text* > *Generate* Gen". Caso a geração de código ocorra corretamente, uma mensagem de sucesso será exibida, similar à área destacada "A". O nome do arquivo gerado é definido no componente `Tag`, propriedade `File`, neste caso o nome do arquivo é `tag_1.tag`. Na Figura 32, é exibida esta informação logo ao lado direito do ícone do componente.

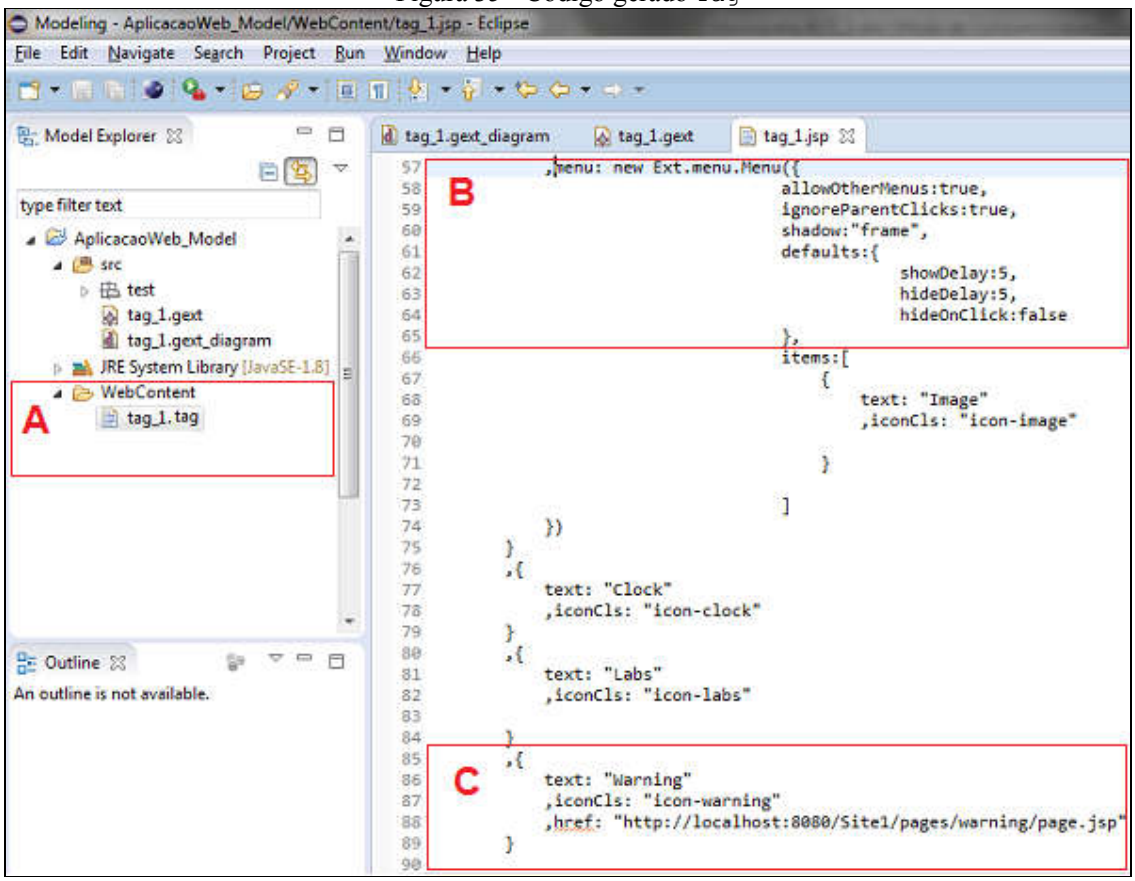
Figura 34 - Execução do gerador de código



Após a execução do gerador de código, o resultado é apresentado na Figura 35. O arquivo é inserido na pasta WebContent do projeto, área destacada "A". O código gerado

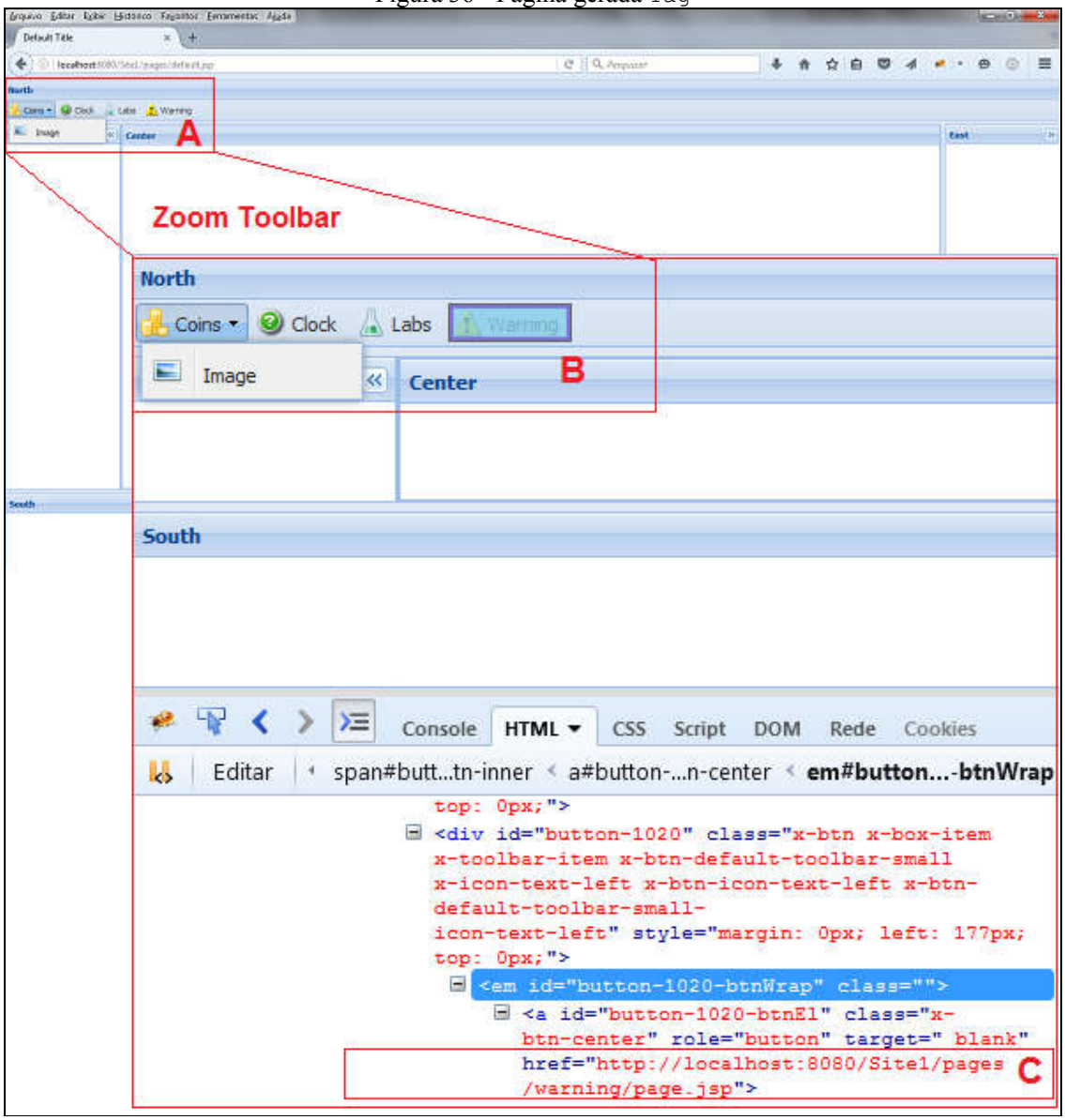
possui alguns trechos de código padrões, na área destacada "B", assim como as propriedades definidas no diagrama, na área destacada "C".

Figura 35 - Código gerado `Tag`



O resultado obtido após a modelagem, inclusão da página gerada no projeto *web* e acesso ao arquivo via *browser*, pode ser visualizado na Figura 36. Pode-se observar na área destacada "A" a barra de navegação do `Toolbar` gerado. Na área "B" é apresentado um *zoom* dos componentes da barra de navegação. Já na área "C" é mostrado o elemento HTML `href` criado com o *link* inserido no modelo. Dessa forma, o código gerado com o Gext JS voltado para o Ext JS foi gerado conforme esperado.

Figura 36 - Página gerada `Tag`



Após a modelagem da `Tag`, modela-se a `Page`. Para fins ilustrativos, é apresentada uma página *web* de um simples cadastro de pessoa. Essa página contém os itens:

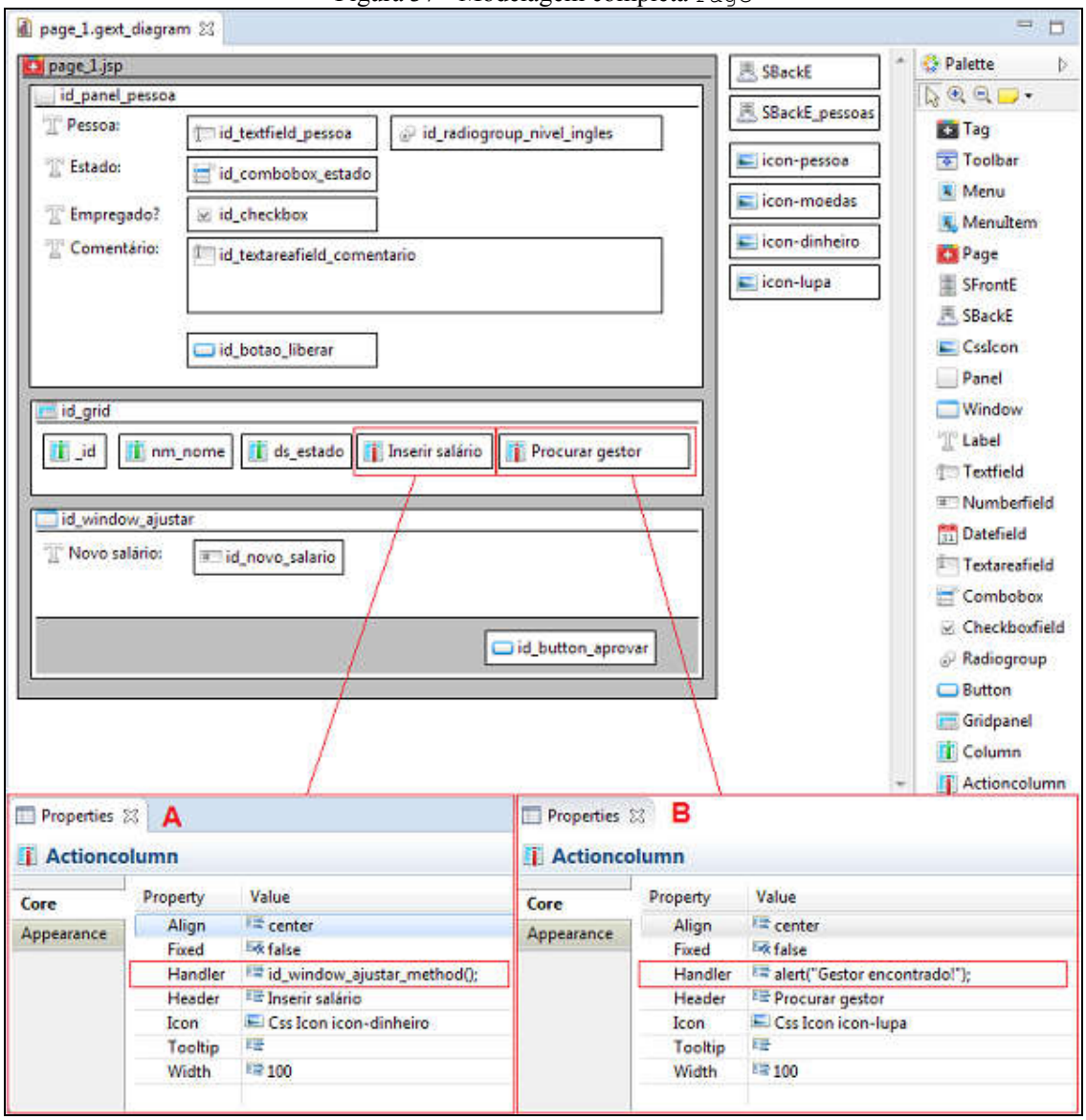
a) `Panel`: um formulário com os campos Nome (`Textfield`), Nível de inglês (`Radiogroup`), Estado (`Combobox`), Empregado (`Checkboxfield`), Comentário (`Textareafield`) e um botão Liberar pessoa (`Button`);

b) `Gridpanel`: uma tabela com as colunas Código(`Column`), Pessoa (`Column`), Estado (`Column`), Inserir salário (`Actioncolumn`), Procurar gestor (`Actioncolumn`);

c) `Window`: uma janela com outro formulário que contém o campo Novo salário (`Numberfield`) e um botão Aprovar salário (`Button`).

Para a criação do arquivo `page_1.gext` e `page_1.gext_diagram`, deve-se seguir os passos descritos anteriormente na Figura 30. Após toda a modelagem, o resultado obtido é apresentado na Figura 37, a área destacada "A" exibe a configuração do evento de clique da célula no componente `Gridpanel`. Este evento invocará o método de criação da `Window`, por meio do método em JavaScript `id_window_ajustar_method()`. A área destacada "B" apresenta o mesmo comportamento, porém apenas invocará a função padrão de alerta do JavaScript.

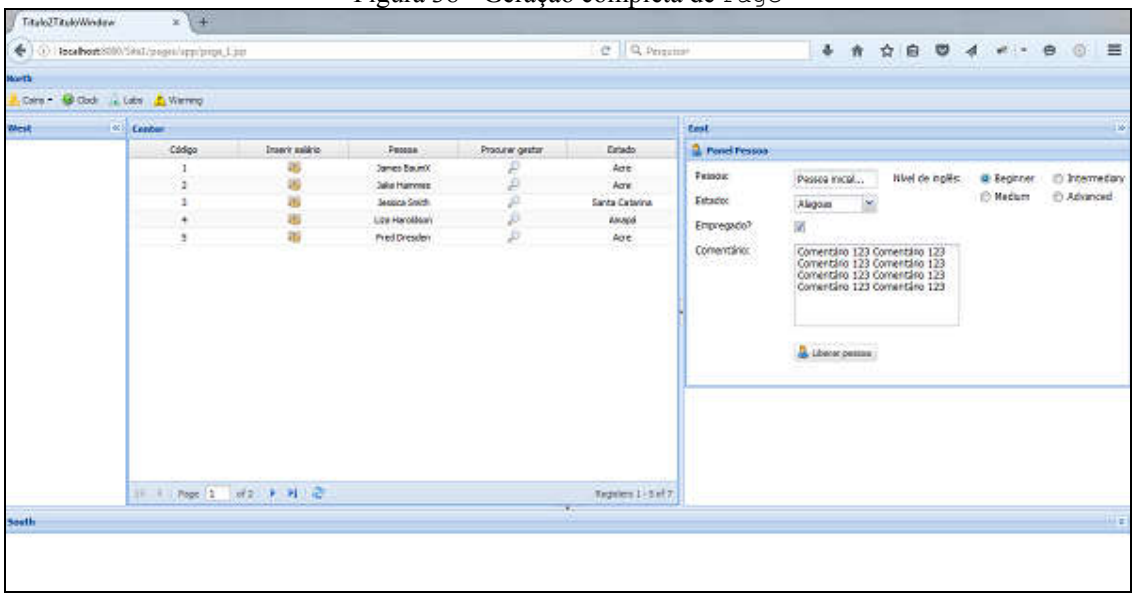
Figura 37 - Modelagem completa `Page`



Ao final da geração e inclusão no arquivo gerado na pasta correspondente do projeto *web*, e acesso ao arquivo via *browser* o resultado obtido é apresentado na Figura 38. Observa-

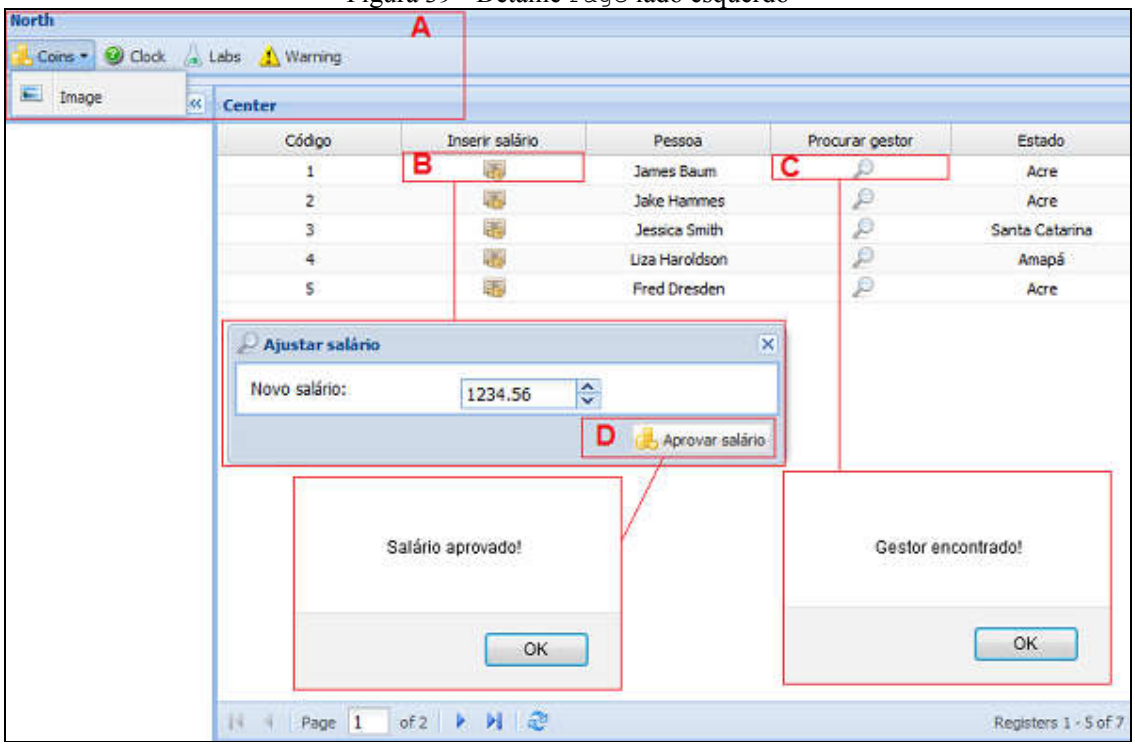
se a página inteiramente gerada conforme os modelos `tag_1.gext` e `page_1.gext`. Para melhor visualização, será apresentado um *zoom* desta página.

Figura 38 - Geração completa de `Page`



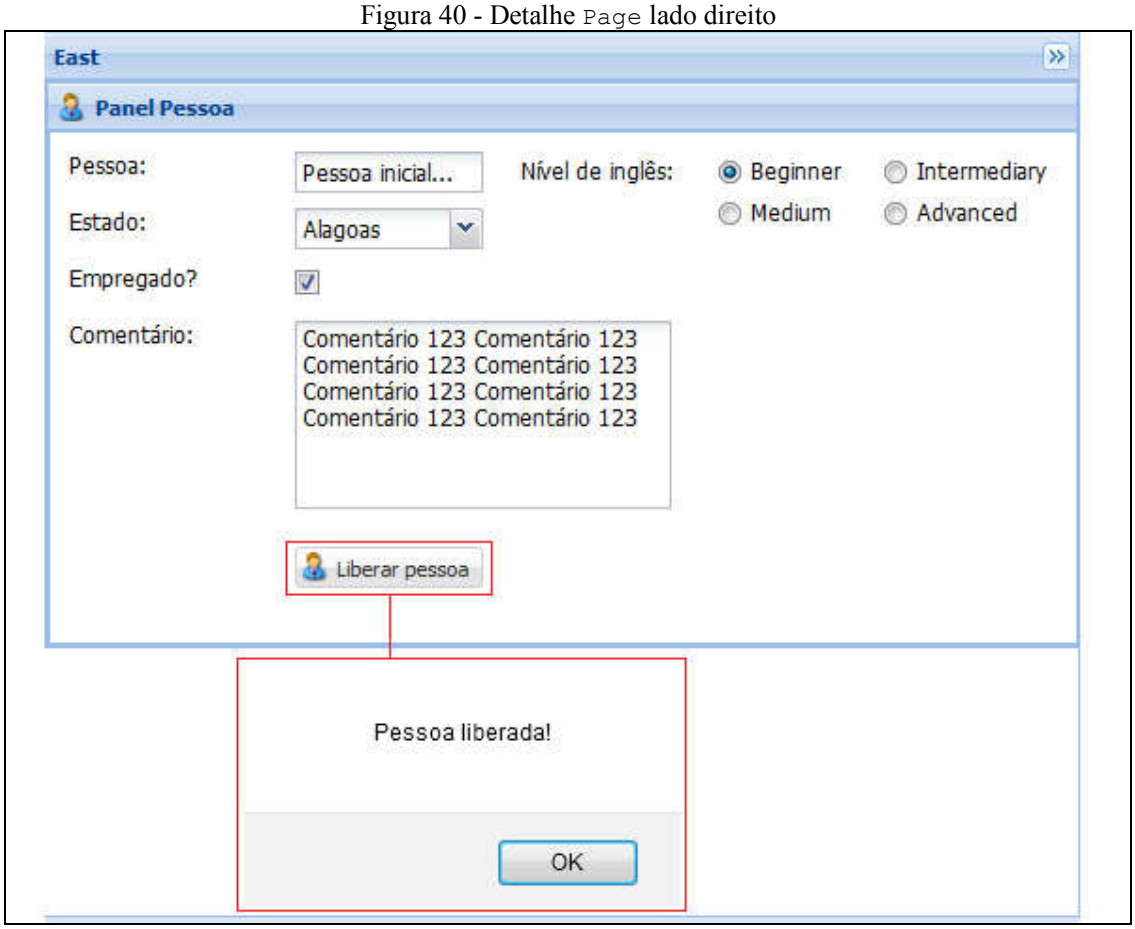
Na Figura 39 é apresentado um *zoom* para o lado esquerdo de `Page`. Observa-se que `Tag` foi referenciada em `Page` para utilizar-se do *layout* padrão anteriormente definido. A área destacada "A" exibe o `Toolbar`. A área destacada "B" exibe a ação de abrir a `Window`, ao pressionar essa célula. As áreas destacadas "C" e "D" exibem suas respectivas mensagens de alerta do JavaScript ao pressionar essas células.

Figura 39 - Detalhe `Page` lado esquerdo



Na Figura 40, destaca-se o lado direito da página apresentada anteriormente. Nela observa-se o resultado final do `Panel` com o formulário e os componentes criados, descritos anteriormente.

Figura 40 - Detalhe `Page` lado direito



3.4 RESULTADOS E DISCUSSÕES

Neste trabalho foi apresentado o desenvolvimento do Gext JS, uma ferramenta visual que facilita o uso do *framework* Ext JS. Os resultados obtidos alcançaram os objetivos propostos. Além de ser possível modelar uma página *web*, é possível gerar o código fonte desta modelagem.

Conforme diversos autores, a programação visual facilita o desenvolvimento de software. Baseando-se no uso de componentes gráficos para abstrair etapas da programação. Utilizar ferramentas de código livre, podem causar um impacto positivo financeira e educacionalmente, pois não existem essas limitações para iniciar o uso deste tipo de ferramenta. A geração de código, aliado ao reúso, garante produtividade no desenvolvimento de software. Dessa forma, unindo esses paradigmas, a curva de aprendizado para o uso de uma ferramenta visual, de código livre e geradora de código é menor se comparado ao ato de iniciar o desenvolvimento de uma ferramenta que não possui essas propriedades.

O Gext JS foi definido desde sua conceituação como um *plugin* da IDE Eclipse. Este objetivo além de ter sido alcançado, teve esse propósito pois o Eclipse já é amplamente

conhecido para o desenvolvimento de software, principalmente para a *web*. Portanto, os desenvolvedores que já estão interados com essa IDE podem facilmente instalar os *plugins* necessários para o Gext JS e utilizar a ferramenta. Sua facilidade de uso também está relacionada ao fato de possuir várias similaridades com o código fonte do Ext JS. Além disso, após a geração de código, o usuário tem a possibilidade de continuar o desenvolvimento via edição do código fonte utilizando os recursos do Eclipse.

O uso do EMF, GMF e do Acceleo foram fundamentais para o desenvolvimento do Gext JS. Por serem oriundos do Eclipse, houve uma fácil aderência entre todos os *plugins*. Isso reduziu os problemas de comunicação entre os *plugins* do Eclipse e do Gext JS. Apesar desta facilidade de integração, foi necessário possuir um bom conhecimento de todas as ferramentas utilizadas para o desenvolvimento, caso contrário provavelmente em algum momento a implementação teria problemas na comunicação entre os *plugins* gerados.

Apesar dos objetivos terem sido alcançados, a geração de código também está relacionada com as desvantagens da MDE. Por exemplo, modelos abstratos são bons para entender um software, porém o gerador de código utiliza implementações genéricas que podem ser eficientes para uma determinada situação (como o uso de componente genérico), porém, pode não ser para outra (como o uso de um componente que necessita ser implementado, pois pode possuir muitas especificidades). Dessa forma, o código fonte criado é limitado aos conhecimentos do programador do gerador de código.

Em relação aos trabalhos correlatos, o Gext JS é o único que integra com a IDE Eclipse. Por outro lado, Sencha Architect, Genexus e o protótipo de Battisti (2014) não dependem de IDE de terceiros para serem utilizados. Especificamente para o SA é necessário ter outras tecnologias instaladas para que seja utilizado. Por se tratarem de propostas acadêmicas, o Gext JS e o protótipo Elicitar de Battisti (2014) não possuem restrições quanto ao seu uso comercial. Todas as ferramentas usam *templates* para geração de código direta ou indiretamente. Apenas o Genexus pode gerar código para mais de uma linguagens de programação. Todas as ferramentas possuem geração de código para ao menos a plataforma *web*. Apenas o protótipo Elicitar de Battisti (2014) faz o processamento de linguagem natural. Contudo, o Genexus é capaz de gerar o *back-end*, pois trata-se de uma ferramenta completa de geração de aplicação. O Quadro 3 apresenta estas informações de forma resumida.

Quadro 3 - Comparação com os trabalhos correlatos

característica	Gext JS	Sencha Architect	Genexus	Battisti (2014)
Integra com a IDE Eclipse	X			
Independente de IDE		X	X	X
Permite uso comercial gratuito	X			X
Utiliza *template* para geração de código	X	X	X	X
Gera código para mais de uma linguagem			X	
Plataforma web	X	X	X	X
Gera o *back-end*			X	
Processamento de linguagem natural				X

4 CONCLUSÕES

Atualmente existem diversas ferramentas e técnicas que facilitam o desenvolvimento de software. Em especial ambientes de desenvolvimento visuais e geradores de código são alguns exemplos. Tais ferramentas são importantes para promover a produtividade ao desenvolver. Com o auxílio de *frameworks*, DSLs, MDE e geradores de código é possível criar um ambiente visual de desenvolvimento de acordo com a necessidade do usuário. Para isso, é importante definir o escopo dessas técnicas e em qual momento serão aplicadas.

Tais técnicas cumpriram exatamente o seu papel no âmbito de conceituação, assim como na parte de implementação. O Ext JS serviu como a linguagem alvo a ser gerada, ele está presente no escopo de *frameworks*. O EMF e GMF foram importantes para a criação da DSL, pois o domínio a ser tratado foi previamente definido ao analisar os componentes a serem selecionados para a geração de código. O EMF e GMF também foram importantes para a concepção da MDE, sendo responsáveis por modelarem a estrutura dos componentes e os relacionamentos entre eles. O Acceleo foi utilizado na etapa de geração de código. Essa geração de código foi possível graças aos modelos dos componentes previamente estruturados.

O objetivo geral referente à criação de um editor gráfico para desenvolvimento visual de páginas baseadas no *framework* Ext JS foi atendido. Este objetivo é a própria ferramenta Gext JS, pois é possível modelar componentes do Ext JS e gerar o código destes modelos. O objetivo específico de criar uma GUI para edição de determinados componentes do Ext JS também foi atendido. Este item é representado pela seção 3.3.1.1. O objetivo específico de disponibilizar um gerador de código fonte que cria o código em Ext JS a partir dos modelos criados na GUI também foi atendido. Este item é representado pela seção 3.3.1.2. Como resultado desses dois objetivos específicos o objetivo geral é alcançado, representado na seção 3.3.2.

Como vantagens o Gext JS foi desenvolvido totalmente com *plugins* oriundos da IDE Eclipse e seu uso ocorre exportando-se como um *plugin* para a IDE Eclipse. Sendo assim, a chance de ocorrer problemas de compatibilidade é reduzida. Além disso, a sua instalação é simples, ou seja, basta incluir os *plugins* descritos na seção 3.3.1. Outro destaque é o fato de que o Gext JS é de uso livre, sendo assim, pode ser utilizado comercialmente em qualquer momento para gerar as páginas *web* que fazem uso do Ext JS, desde que possua aderência ao projeto existente. Além de poder receber manutenções de outros desenvolvedores, pois seu código é livre.

Como desvantagens, o Gext JS gera apenas o *front-end*. Sendo assim, deve-se desenvolver a estrutura do servidor, caso haja alguma requisição do tipo *Hypertext Transfer Protocol* (HTTP). Outra desvantagem trata-se do código gerado estar amarrado à tecnologia JSP, pois alguns componentes, como `Tag`, tiveram seu *template* escrito com elementos do JSP. Visualmente, o editor gráfico do Gext JS não apresenta exatamente a aparência da página que está sendo criada, mas somente um esquema dos elementos que a compõe. Apesar disso, os ícones utilizados para representar os componentes de formulário são similares aos próprios componentes de formulário. Isso facilita a identificação visual por parte do usuário.

Sendo assim, apesar das desvantagens, confirmou-se que é possível criar um editor visual gratuito voltado para a linguagem Ext JS e gerar seu código fonte. Conclui-se que o Gext JS é um bom caso de estudo para as técnicas relacionadas à *frameworks*, MDE, DSL e geração de código. De fato, ele não é superior às ferramentas comerciais já consolidadas no mercado, mas sem dúvida pode ser utilizado e melhorado para atender a uma necessidade específica.

4.1 EXTENSÕES

O trabalho desenvolvido alcançou os objetivos esperados. Porém, sugere-se as seguintes extensões:

a) incluir outros componentes gráficos do Ext JS no Gext JS, por exemplo o `Base`, `Display`, `File`, `Hidden`, `HtmlEditor`, `Picker`, `Tabpanel` entre outros, para oferecer mais componentes ao usuário;
b) adicionar ao Gext JS a capacidade de gerar código para outras ferramentas baseadas em JavaScript, por exemplo o Angular JS e Dojo Toolkit, para possuir mais opções de geração de código em JavaScript;
c) melhorar o desenvolvimento visual, pois a disposição dos componentes no editor gráfico não é exatamente igual às propriedades configuradas na aba *Properties*, como por exemplo a posição `x` e `y` de um componente no painel do editor. Isso facilitará a interação com o usuário;
d) adicionar ao Gext JS a capacidade de gerar o código para o *back-end*, por exemplo, as operações básicas de inserir, editar e excluir registros, permitindo o desenvolvedor abstrair este processo.

REFERÊNCIAS

ABI-ANTOUN, Marwan. Making frameworks work: a project retrospective. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS , 7th, 2007, Montreal. **Proceedings…** New York: ACM, 2007, p. 1004-1018. Disponível em: <www.cs.wayne.edu/~mabianto/papers/07-oopslaexp.pdf>. Acesso em: 05 abr. 2015.

ALMEIDA, Eduardo S. et al. A Systematic Approach to Design Domain-Specific Software Architectures. **Journal of Software**, São Paulo, v. 2, n. 2, p. 38–51, Ago. 2007. Disponível em:<www.researchgate.net/publication/42804451_A_Systematic_Approach_to_Design_Domain-Specific_Software_Architectures>. Acesso em: 20 mar. 2016.

ARTECH CONSULTORES SRL. **Genexus**. [S.l.], 2015. Disponível em: <www.genexus.com/produtos/genexus?pt>. Acesso em: 22 mar. 2015.

BATTISTI, Leandro V. **Elicitar:** protótipo de gerador de código a partir de especificação com padrões de requisitos. 2014. 54 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <dsc.inf.furb.br/arquivos/tccs/monografias/2014_1_leandro-vilson-battisti_monografia.pdf>. Acesso em: 25 abr. 2015.

BAUER, Veronika. Facts and fallacies of reuse in practice. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 17th, 2013, Genova. **Proceedings**… Washington: IEEE Computer Society, 2013. p. 431-434. Disponível em: <www4.informatik.tu-muenchen.de/~bauerv/docs/bauer2013docsym.pdf>. Acesso em: 01 maio 2015.

CHAVES, Helder R. O. **Aplicação web para monitorar frequência, notas e avaliações, a partir da ferramenta Genexus**. 2003. 54 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <dsc.inf.furb.br/arquivos/tccs/monografias/2003-1heldechavesvf.pdf>. Acesso em: 10 out. 2015.

DEURSEN, Arie V.; KLINT, Paul. Domain-Specific Language design requires feature descriptions. **Journal of Computing and Information Technology - CIT**, Amsterdan, v. 10, n. 2, p. 1–17, 2002. Disponível em: <hrcak.srce.hr/file/69420>. Acesso em: 04 abr. 2015.

FANG, Walter; SO, Andrew; TKACH, Daniel. **Visual modeling technique**. Menlo Park: Addison Wesley Longman, 1996.

FAYAD, Mohamed E.; JOHNSON, Ralph E. **Domain specific application frameworks:** frameworks experience by industry. New York: Wiley, 1999.

FAYAD, Mohamed. E.; SCHMIDT, Douglas C. Object-oriented application frameworks. **Communications of the ACM**, [S.l.], v. 40, n. 10, p. 32-38, Oct. 1997. Disponível em: <www.dre.vanderbilt.edu/~schmidt/CACM-frameworks.html>. Acesso em: 05 abr. 2015.

FRANCE, Robert; RUMPE, Bernhard. Model-Driven development of complex software: a research roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING -FUTURE OF SOFTWARE ENGINEERING, 29th, 2007, Minneapolis. **Proceedings…**Washington: IEEE Computer Society, 2007. p. 37–54. Disponível em: <www.arxiv.org/ftp/arxiv/papers/1409/1409.6620.pdf >. Acesso em: 16 maio 2015.

GRONBACK, Richard C. **Eclipse modeling project:** a domain-specific language (DSL) toolkit. New York: Addison-Wesley, 2009.

HERRINGTON, Jack. **Code generation in action**. Greenwich: Manning, 2003.

HIRAMA, Keichi. **Engenharia de software:** qualidade e produtividade com tecnologia. Rio de Janeiro: Elsevier, 2011.

HORSTMANN, Cay. **Padrões e projeto orientados a objetos**. 2. ed. Tradução Bernardo Copstein. Porto Alegre: Bookman, 2007.

KIRK, Douglas et al. Identifying and addressing problems in object-oriented framework reuse. **Empirical Software Engineering**, Glasgow, v. 12, n. 3, p. 243–274, Jun. 2007. Disponível em: <link.springer.com/article/10.1007/s10664-006-9027-z>. Acesso em: 15 mar. 2015.

LOPES, Sérgio F. et al. Framework characteristics: a starting point for addressing reuse difficulties. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, 4th, 2009, Guimarães, Portugal. **Proceedings…**Porto: The Fourth International Conference on Software Engineering Advances, 2009. p. 256–264. Disponível em: <ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5298192>. Acesso em: 16 maio 2015.

ORACLE. **NetBeans**. [S.l.], 2015. Disponível em: <netbeans.org>. Acesso em: 22 mar. 2015.

SENCHA. **Ext JS**. [S.l.], 2015a. Disponível em: <www.sencha.com/products/extjs>. Acesso em: 15 mar. 2015.

_____, **Sencha architect**. [S.l.], 2015b. Disponível em: <docs.sencha.com/architect/3/getting_started/installation_setup.html>. Acesso em: 15 mar. 2015.

SILVEIRA, Janira. **Extensão da ferramenta Delphi2Java-II para suportar componentes de banco de dados**. 2006. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <campeche.inf.furb.br/tccs/2006-I/2006-1janirasilveiravf.pdf>. Acesso em: 20 mar. 2015.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

SRINIVASAN, S. Design patterns in object-oriented frameworks. **ACM Computer**, Los Alamitos, v. 32, n.2, p. 24-32, Feb. 1999. Disponível em: <www.panda.sys.t.u-tokyo.ac.jp/kushiro/ReferencePaper/06GUIpaper/00745717.pdf>. Acesso em: 16 maio 2015.

THE ECLIPSE FOUNDATION. **Acceleo**. [S.l.], 2015a. Disponível em: <wiki.eclipse.org/Acceleo/Getting_Started>. Acesso em: 22 mar. 2015.

_____. **Graphical modeling project**. [S.l.], 2015b. Disponível em: <eclipse.org/modeling>. Acesso em: 10 mar. 2015.

VIANA, Matheus C. **Desenvolvimento e reúso de frameworks com base nas características do domínio**. 2014. 212 f. Tese (Doutorado em Ciência da Computação) - Universidade Federal de São Carlos, São Carlos. Disponível em: <www.bdtd.ufscar.br/htdocs/tedeSimplificado//tde_busca/arquivo.php?codArquivo=7636>. Acesso em: 22 mar. 2015.

VOELTER, Markus. **DSL engineering**: design, implementing and using domain-specific languages. [S.l.]: [s.n.], 2013. Disponível em: <voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf>. Acesso em: 05 abr. 2015.

XU, Lugang; BUTLER, Gregory. Cascaded refactoring for framework development and evolution. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE, 06th, 2006, Sydney. **Proceedings…** Washington: IEEE Computer Society, 2006. p. 1-10. Disponível em: <users.encs.concordia.ca/~gregb/home/PS/lxu-phd-thesis.pdf>. Acesso em: 04 abr. 2015.

APÊNDICE A – Relação dos casos de uso

Este apêndice contém o detalhamento dos casos de uso apresentados na Figura 8 da seção 3.2.1. O Quadro 4 apresenta o caso de uso UC01. O Quadro 5 apresenta o caso de uso UC02. O Quadro 6 o caso de uso UC03.

Quadro 4 - Detalhamento do caso de uso UC01 Modelar página *web*

UC01 - Modelar página *web*: permite a criação de modelos via paleta de componentes.	
Pré-condições	Para testar o código gerado é necessário ter um projeto *web* previamente configurado com o Ext JS. Este projeto deve conter uma pasta chamada WebContent.
Cenário principal	01) O usuário inicia a modelagem criando um novo modelo de página *web*. 02) O usuário modela a página *web* por meio da seleção de componentes na paleta de componentes. 03) O usuário edita as propriedades destes componentes. 04) O usuário vincula os componentes entre si. 05) O Gext JS armazena o modelo criado.
Fluxo alternativo 01	01.01) No passo 02 o usuário pode editar um modelo anteriormente criado.
Pós-condições	Nenhuma.

Quadro 5 - Detalhamento do caso de uso UC02 Validar modelo

UC02 - Validar modelo: permite a validação de consistência de modelos.	
Pré-condições	Possuir um modelo a ser validado
Cenário principal	01) O usuário executa o validador de modelos. 02) Uma mensagem de sucesso ou de erro é apresentada de acordo com a validação do modelo. 03) Caso ocorra erro, os componentes configurados incorretamente serão exibidos com uma sugestão de correção. 04) A modelagem pode prosseguir independente de haver erro, ou não.
Fluxo alternativo 01	01.01) No passo 01 o usuário pode não executar o validador e nenhuma mensagem de validação é apresentada.
Pós-condições	Nenhuma.

Quadro 6 - Detalhamento do caso de uso UC03 Gerar código da página *web*

UC03 - Gerar código da página *web*: permite a geração da página *web* a partir do modelo.	
Pré-condições	Possuir um modelo para gerar a página *web*.
Cenário principal	01) O usuário aciona o gerador de código. 02) O Gext JS faz a geração de código criando um arquivo na pasta WebContent. 03) O usuário manipula o arquivo gerado de acordo com sua necessidade.
Fluxo alternativo 01	01.01) No passo 01 o usuário pode optar por não gerar a página *web* e manter apenas os modelos.
Pós-condições	Nenhuma.