UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PAROLA: UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES J2SE BASEADAS EM *PLUG-INS*.

TRABALHO DE GRADUAÇÃO

Daniel Michelon De Carli

Santa Maria, RS, Brasil

2009

PAROLA: UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES J2SE BASEADAS EM *PLUG-INS*.

por

Daniel Michelon De Carli

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas (UFSM)

Trabalho de Graduação Nº 282
Santa Maria, RS, Brasil

2009

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

PAROLA: UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES J2SE BASEADAS EM *PLUG-INS*.

elaborado por
Daniel Michelon De Carli

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Marcos Cordeiro d'Ornellas (UFSM)
(Presidente/Orientador)

Prof^{a}. Dr^{a}. Lisandra Manzoni Fontoura (UFSM)

M^{a}. Gabriela Carla Bauermann

Santa Maria, 16 de julho de 2009.

AGRADECIMENTOS

Agradeço acima de tudo a Deus - inteligência suprema, causa primária de todas as coisas – pela vida.

Agradeço à minha família por ser a minha fortaleza nas horas de dificuldade, pela educação moral e por buscar me proporcionar uma educação acadêmica de excelência que, com muito esforço, conseguiram.

Agradeço à minha noiva, Graziele Camargo Kemmerich, pelo carinho, amor e dedicação. Além disso, por ser incansável - aprendendo computação - para me auxiliar na revisão deste trabalho.

Agradeço a meus colegas da Animati, em especial ao Jean Carlo Berni, pela leitura do presente trabalho e pelas diversas sugestões de melhorias no texto.

Agradeço ao meu orientador, por acreditar no meu potencial, bem como pelos momentos de conselho e críticas, que me foram bastante úteis na busca de ser melhor profissional.

Agradeço aos bons espíritos pelo auxílio fraternal, pela indução de boas idéias e pela intuição que muito me auxilia.

Agradeço aos meus professores, pelo conhecimento e experiência compartilhada, por todo o suporte concedido.

Agradeço a todos aqueles que estiveram presentes na minha vida e que de uma forma ou de outra contribuíram para a consolidação dessa etapa.

‘A pessoa é, acima de tudo, a sua mente. O que elabora, torna-se; quanto cultiva, experimenta.’

— JOANNA DE ÂNGELIS

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

PAROLA: UM *FRAMEWORK* PARA CONSTRUÇÃO DE APLICAÇÕES J2SE BASEADAS EM *PLUG-INS*.
Autor: Daniel Michelon De Carli
Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas (UFSM)
Local e data da defesa: Santa Maria, 16 de julho de 2009.

O Parola é um *framework* para construção de aplicações de processamento de imagens que atua na construção da interface gráfica do usuário, controlando os métodos matemáticos e gerenciando as imagens. Os sistemas, que fazem uso da abordagem proposta pelo Parola, recebem novas funcionalidades através do desenvolvimento e inserção de *plug-ins*. Os *plug-ins* são desenvolvidos com maior independência estrutural que os componentes de *software* convencionais pois, necessitam conhecer apenas as interfaces dos objetos de fronteira para se comunicarem com o sistema. Dessa forma, o desenvolvedor do *plug-ins* não precisa conhecer em profundidade a arquitetura do sistema, mas sim as regras básicas de comunicação entre a plataforma de *software* e o *plug-in*. Esse trabalho busca demonstrar o uso de técnicas de engenharia de *software* e padrões de projeto para a construção de um *framework*, que implementa o gerenciamento de *plug-ins*, usando a tecnologia J2SE (Java 2 *Platform, Standard Edition*).

Palavras-chave: Frameworks, plug-ins, padrões de projeto, processamento de imagens.

ABSTRACT

Graduation Work
Graduate Program in Computer Science
Federal University of Santa Maria

PAROLA: A FRAMEWORK FOR CONSTRUCTING J2SE APPLICATIONS BASED ON PLUG-INS.
Author: Daniel Michelon De Carli
Advisor: Prof. Dr. Marcos Cordeiro d'Ornellas (UFSM)

Parola is a framework for image processing applications that works by constructing graphical user interfaces, controlling the mathematical methods and managing images. Systems built by the Parola approach receive new features by developing and installing plug-ins. Plug-ins are created with more structural independence than traditional software components, because they just need to know the border objects to communicate with the system. Thus, the plug-ins' developers don't need profound knowledge regarding the system's architecture, but they do need to know the basic rules for communication between the software platform and the plug-in. This work demonstrates the use of some software engineering techniques and design patterns to construct a framework for managing plug-ins using the J2SE (Java 2 Platform, Standard Edition) technology.

Keywords: Frameworks, plug-ins, design patterns, image processing.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BCE	*Boundary, Control and Entity*
CUDA	*Compute Unified Device Architecture*
DAO	*Data Access Object*
GPU	*Graphics Processing Unit*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
ISO	*International Organization for Standardization*
JVM	*Java Virtual Machine*
JAI	*Java Advanced Imaging*
JNI	*Java Native Interface*
J2SE	*Java 2 Platform Standard Edition*
JSE	*Java Platform Standard Edition*
SGML	*Standard Generalized Markup Language*
SOAP	*Simple Object Access Protocol*.
W3C	*The World Wide Web Consortium*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A elaboração de sistemas computacionais vem aumentando seu grau de complexidade na medida que novos componentes passam a ser inter-relacionados para geração de aplicações. Percebe-se que a reutilização de código é uma questão fundamental para a criação de sistemas de forma mais rápida e eficiente. Tanto na indústria, como em laboratórios de pesquisa em universidades, o fator tempo (ou *time-to-market*) se caracteriza como diferencial e indicador de produtividade. Concernente a isso, entregar sistemas com alto valor agregado e com diferenciais competitivos é o objetivo almejado por qualquer organização.

Produzir *software* com rapidez e eficácia é uma atividade complexa, que exige um grande esforço para o seu projeto e sua implementação. Entretanto, o que geralmente ocorre é a redescoberta e reinvenção dos aspectos centrais da aplicação, tornando o desenvolvimento do *software* custoso e de baixa qualidade. Quando esse processo ocorre o mesmo é chamado de reescrever o código (SCHMIDT; GOKHALE; NATARAJAN, 2004). Somando-se a isso, conforme Freitas (FREITAS, 2006), podem ocorrer situações onde a falta de planejamento conduza à implementação de sistemas monolíticos, os quais apresentam uma grande quantidade de compartilhamento de dados, de variáveis globais e de um fluxo de controle caótico. Sistemas monolíticos grandes costumam ser de difícil compreensão e, consequentemente, as atividades de manutenção transforma-se em atividades desafiadoras e normalmente são onerosas.

Para minimizar os efeitos dessa complexidade, é comum a utilização de técnicas para permitir a escalabilidade no desenvolvimento dos sistemas. Uma forma de se facilitar o crescimento do *software* é estruturá-lo para que este aceite a inserção de novas funcionalidades conforme a demanda. Para tanto, organizar o sistema como *framework* torna-se uma opção racional. Para Fayad (FAYAD, 2001), um *framework* é uma coleção de componentes implementados completamente ou parcialmente com padrões predefinidos de

cooperação entre eles. A arquitetura de um *framework* corresponde à definição de seus módulos e de como a interação entre estes é realizada.

Frameworks orientados a componentes (ou objetos) tornaram-se populares na indústria de *software* durante os anos noventa. Numerosos *frameworks* foram desenvolvidos por empresas e universidades para diversos domínios de aplicação, incluindo interfaces gráficas de usuário (por exemplo Java *Swing* e outras bibliotecas Java, *Microsoft MFC*), editores gráficos (*HotDraw, Stingray's Objective Views*), aplicações de negócios (IBM *San Francisco*), servidores *network* (Java *Jeeves*), dentre outros. Quando combinados entre si, os *frameworks* fornecem uma poderosa solução para reuso de *software* em larga escala (FAYAD; HAMU, 1999; HAMU, 1999) apud (FAYAD, 2001). Pode-se firmar que os *frameworks* são ferramentas que auxiliam a construção de aplicações através do reuso de estruturas comuns a vários domínios de problemas.

Além disso, a questão de foco e objetivo, aliada a ambientes com alta rotatividade de pessoas se tornam um problema para a construção de sistemas de forma contínua. Nas universidades, grandes projetos de *software* são uma tarefa desafiadora, tendo em vista que a mão-de-obra normalmente empregada é composta por alunos de graduação e mestrado, que costumam passar um período relativamente curto na instituição. Alunos de graduação atuam de forma mais substancial em projetos de *software* quando estes representam seu trabalho de conclusão de curso. No entanto, os códigos fonte produzidos geralmente são mal documentados, o que torna o processo de reutilização de componentes complexo ou impraticável. Mestrandos, por sua vez, possuem em média dois anos para o desenvolvimento das suas atividades acadêmicas e pesquisas científicas, e são detentores de diversas responsabilidades como docência orientada e a publicação de seus estudos.

Laboratórios de pesquisa que buscam ter um desenvolvimento de *software* continuado normalmente enfrentam o panorama citado. Com o objetivo de minimizar o prejuízo causado pelo cenário, propõe-se uma abordagem baseada em *plug-ins*. *Plug-ins* são componentes de *software* que apresentam sua arquitetura independente da aplicação principal, pois se interrelacionam com o sistema através dos objetos de fronteira. Pode-se destacar os principais pontos positivos dessa abordagem: Primeiramente, *plug-ins* mal projetados não comprometem todo o sistema. Em segundo lugar, é mais fácil reescrever um *plug-in* do que uma aplicação inteira.

O Parola busca ser um *framework* que auxilia no gerenciamento de métodos de pro-

cessamento e análise de imagens digitais que são desenvolvidos por demanda. Para isso, baseia-se em padrões de projeto, utilizando técnicas de engenharia de *software* para implementar uma arquitetura baseada em *plug-ins* de maneira consistente.

O presente trabalho de graduação segue a seguinte organização: o capítulo 2 apresenta uma revisão bibliográfica a respeito de *frameworks*, sistemas baseados em componentes e *plug-ins*, padrões de projeto e Java. O capítulo 3 descreve o processo de desenvolvimento da ferramenta, abordando considerações de *design*, arquitetura e implementação, como também a forma de construir *plug-ins* para o Parola. O capítulo 4 expõe os resultados obtidos. Por fim, o capítulo 5 apresenta as conclusões.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo apresenta-se uma revisão bibliográfica sobre os tópicos relacionados a este trabalho. Primeiramente serão mostradas questões referentes à *frameworks*. Será tratado, também, o desenvolvimento baseado em componentes e *plug-ins*. A seguir, é apresentada a temática de Padrões de Projeto, sendo explicado os principais padrões que foram utilizados para o desenvolvimento do *framework* Parola. Além disso, apresentam-se questões referentes à linguagem de programação Java, bem como sobre a plataforma J2SE (Java 2 *Platform, Standard Edition*).

2.1 *Frameworks*

Parnas (PARNAS; CLEMENTS; WEISS, 1985) afirma que um *framework* implementa a arquitetura de *software* para uma família de aplicações com características similares. Pode-se, também, definir *framework* como sendo um conjunto de classes que cooperam entre si e constroem um projeto reutilizável de *software*. Além disso, um *framework* captura as decisões e projetos que são comuns ao seu domínio de aplicação (GAMMA et al., 2005). Nesse caso, *Frameworks* são mais aplicáveis aos domínios de problemas onde exista uma grande uniformidade em termos de funcionalidade e requisitos (SCHMIDT; GOKHALE; NATARAJAN, 2004). Um *framework* tem como uma das funções principais permitir que módulos ou funcionalidades sejam adicionados por demanda, aumentando seu escopo de utilização.

A padronização da estrutura da aplicação permite uma redução significativa do tamanho e complexidade do código fonte que deve ser escrito por desenvolvedores, os quais customizam o *framework* (FAYAD, 2001). Os esforços de desenvolvimento do *framework* devem ser direcionados no sentido de uma modelagem que tente prever sua escalabilidade, uma vez que sua meta é abranger o maior número de funcionalidades para determinado

domínio de aplicação.

Em um *framework*, componentes ou módulos são chamados de pontos de variação ou *hotspots* (PREE, 1999, 1995). Uma aplicação criada a partir do *framework* pode ser "customizada" através dos *hotspots*, que representam pontos de adaptação de código que podem ser redefinidos (FAYAD; SCHMIDT; JOHNSON, 1999). As partes fixas do *framework* são chamadas *frozenspots* e representam as estruturas imutáveis do sistemas. Dessa forma, temos:

- *Frozenspots*
  - Partes fixas de um *framework*;
  - Serviços já implementados pelo *framework*;
  - Geralmente realizam chamadas indiretas ou genéricas aos *hotspots*.
- *Hotspots*
  - Partes flexíveis de um *framework*;
  - Pontos extensíveis, para criação de novos serviços/funcionalidades;
  - Partes nos quais os programadores que usam o *framework* adicionam o seu código para especificar uma funcionalidade de sua aplicação;
  - *Hotspots* são geralmente implementados através de herança e de métodos abstratos.

Neste trabalho busca-se desenvolver um *framework* direcionado para o processamento e análise de imagens. Dentro desse propósito a arquitetura do sistema foi elaborada para permitir com que novos métodos de processamento ou análise sejam adicionados ao *framework* de forma simplificada, correspondendo aos *hotspots*. A parte fixa do sistema equivale a arquitetura e a forma como os seus componentes se inter-relacionam. Também existem componentes que são aplicáveis à maioria dos métodos, como por exemplo, módulos para abrir, salvar e visualizar de imagens, considerados como *frozenspots*.

2.2 Sistemas Baseados em Componentes e *Plug-ins*

Sistemas baseados em componentes, são facilmente personalizados ou adaptados para domínios de aplicações específicos. Aplicações de processamento e análise de imagens

são utilizadas nos mais diversos ramos de pesquisas científicas, onde cada área tem suas características e particularidades. Um sistema de propósito geral necessita, dessa forma, de adequações para ser utilizado de maneira mais eficaz e eficiente.

Da mesma forma que os paradigmas de desenvolvimento estruturado e orientado a objetos influenciaram programadores e projetistas de todo o globo, o modelo de desenvolvimento de *software* baseado em componentes emerge como a atual revolução na concepção de sistemas computacionais (VITHARANA, 2003) apud (WELFER, 2004). Essa tendência é consequência natural de uma série de vantagens como fácil manutenção, maior qualidade e tempo e custos reduzidos de desenvolvimento.

Conforme destaca Welfer (WELFER, 2004), o desenvolvimento baseado em componentes possui três personagens principais: o desenvolvedor do componente, o montador e o cliente ou usuário final. De maneira simplificada, o desenvolvedor é aquele que cria o componente, isto é, o programador; o montador necessita conhecer apenas as interfaces do componente, utilizando-o e combinando com outros componentes para resolução de algum problema computacional; e, por fim, o cliente, ou usuário final, é aquele que recebe uma aplicação a qual foi construída com o advento de componentes previamente selecionados e que, dessa forma, atendem sua lista de requisitos. A Figura 2.1 demonstra a atuação desses três elementos no processo de desenvolvimento de software desde a sua concepção até o seu uso pelo cliente final (WELFER, 2004). Primeiramente, os componentes são fabricados pelo desenvolvedor para vários fins, sendo que não há uma relação explícita entre eles, isto é, são concebidos para serem independentes. Em um segundo momento, o montador precisa encontrar e inter-relacionar os componentes a fim de resolver o problema do cliente. E, por fim, tem-se o produto de software pronto para ser utilizado pelo cliente.

O principal desafio do desenvolvimento de *software* baseado em componentes é o de criar mecanismos para gerenciar a refatoração de código que sempre ocorre quando os requisitos do sistema se modificam de tal forma que sua manutenção seja simples e rápida (BOOCH; JACOBSON; RUMBAUGH, 1999). Para tanto, devem-se administrar artifícios para permitir a personalização do sistema garantindo a elaboração de aplicações direcionadas. Um *framework* facilmente adaptável deverá garantir a elaboração de sistemas cujas demandas sejam individualizadas.

Entretanto, Wolfinger (WOLFINGER, 2008) defende que o desenvolvimento baseado

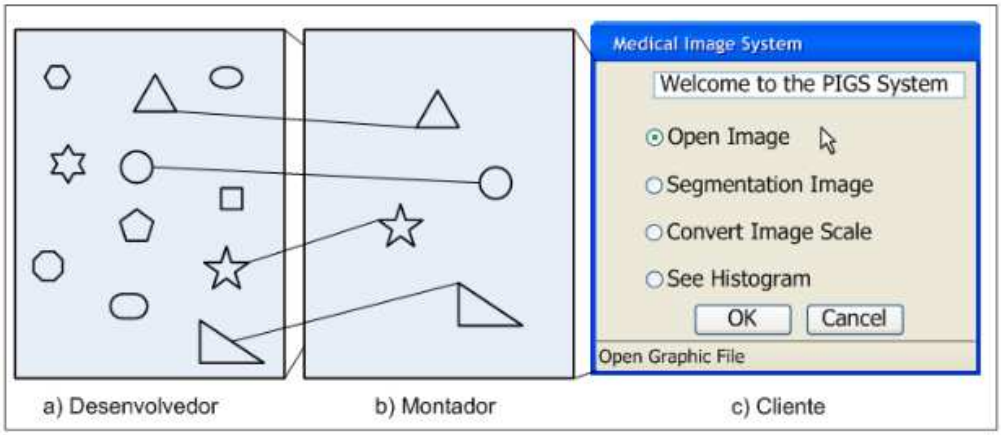

Figura 2.1: Os três personagens do processo de componentização segundo Padmal Vitharana (VITHARANA, 2003) apud (WELFER, 2004).(Fonte WELFER, 2004)

em componentes cria sistemas monolíticos onde as funcionalidades são habilitadas ou não para execução, e que modificações implicam na reposição de grandes partes do *software*. Para resolver esse problema Wolfinger sugere que sistemas baseados em *plug-ins* carregados em tempo de execução e mais independentes da arquitetura do sistema resolvem melhor o problema da complexidade no desenvolvimento de aplicações customizáveis.

O *framework* Parola engloba ambas as soluções. Existem componentes desenvolvidos como objetos, que são comuns a várias funcionalidades do sistemas, como por exemplo, abrir, salvar e visualizar imagens. Esses métodos foram definidos como componentes por questões de padronização e controle da aplicação. Também é possível incluir novos módulos na forma de *plug-ins* que são carregados para a aplicação em tempo de execução. Os *plug-ins* facilitam a customização do sistema dependendo do seu contexto de aplicação e permitem que terceiros possam desenvolver módulos de forma mais independente (WOLFINGER, 2008).

2.3 Padrões de Projeto

A utilização de padrões em projetos de desenvolvimento de *software* com programação orientada a objetos ganhou força com a publicação dos 23 padrões de projeto pela Gangue dos Quatro "GoF"(GAMMA et al., 2005). Esses padrões são divididos em três categorias: os padrões de criação, padrões estruturais e os padrões comportamentais. A primeira categoria representa formas para o processo de criação ou instanciamento de objetos. A segunda identifica os padrões que compõem as classes ou grupos de classes e os padrões comportamentais caracterizam a interação e responsabilidades entre os objetos,

isto é, a comunicação entre eles (COOPER, 1998).

Conforme Deitel (DEITEL; DEITEL, 2004) os padrões de projeto beneficiam os desenvolvedores do sistema das seguintes formas:

1. Ajudando a construir *software* confiável com arquiteturas comprovadas e a experiência acumulada das empresas;
2. Promover a reutilização de projetos em tempos futuros;
3. Ajudar a identificar erros e armadilhas comuns que ocorrem quando se constroem sistemas;
4. Ajudar a projetar sistemas de forma independente da linguagem na qual estes serão implementados;
5. Estabelecer um vocábulo de projeto comum entre desenvolvedores;
6. Encurtar a fase de projeto em um processo de desenvolvimento de *software*.

Na elaboração do *framework* de que trata esse trabalho foram empregados alguns padrões de projetos. Podemos destacar o *Observer* descrito por Gamma (GAMMA et al., 2005) e o *Data Access Object*(DAO), explicado pela Sun(SUN, 2002).

2.3.1 *Observer*

Conforme Gamma (GAMMA et al., 2005), o padão *Observer* tem o objetivo de ser usado quando um objeto muda de estado e é necessário notificar todos os seus dependentes atualizando-os automaticamente.

Pode-se verificar uma representação do padrão *Observer* na Figura 2.2. Se observa que o *Subject* representa uma classe abstrata ou uma interface que oferece mecanismos para adicionar e remover observadores. A implementação da lógica será feita na classe *ConcrecteSubjec* que envia uma notificação para os observadores quando seu estado muda. O *Observer* define uma interface de atualização para objetos que devem ser notificados sobre mudanças em um *Subject*. Já o *ConcrecteObserver* implementa a lógica necessária para executar uma ação quando notificado.

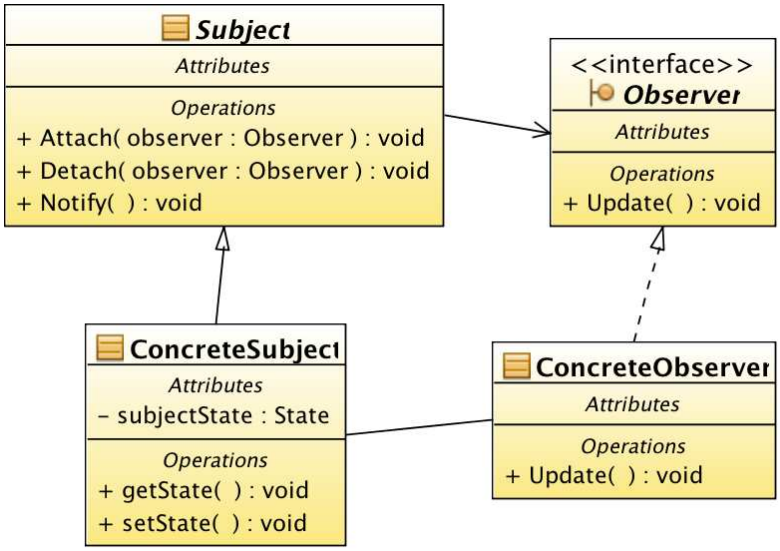

Figura 2.2: Diagrama de classes do padrão *Observer*.

2.3.2 *Data Acess Object* (DAO)

Conforme a Sun (SUN, 2002), o uso de um *Data Access Object* abstrai e encapsula todo acesso aos dados, sendo que o DAO é responsável por gerenciar a conexão com a fonte de dados para obter e armazenar as informações. Pode-se visualizar na Figura 2.3 o diagrama de classes do padrão DAO. O *BusinessObject* é o objeto que requer o acesso à base de dados tanto para obter quanto para armazenar informações. O *DataAccessObject* abstrai o acesso à fonte de dados (*DataSource*) tanto para gravação, quanto para acesso as informações. O *TransferObject* é usado para carregar os dados.

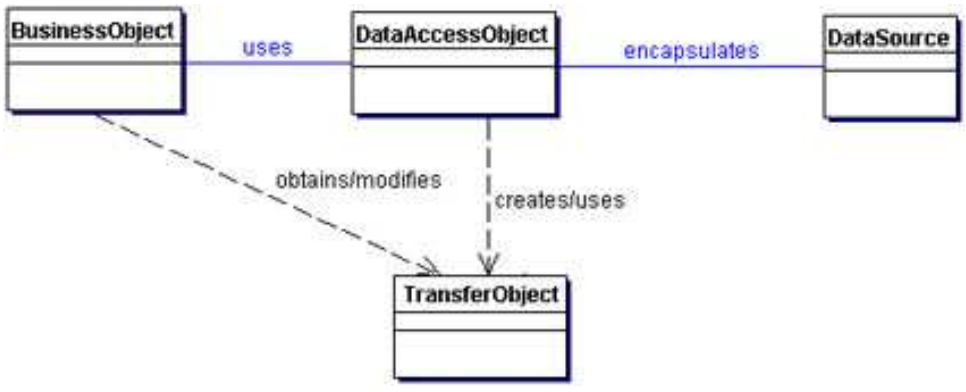

Figura 2.3: Diagrama de Classes do Padrão DAO.(Fonte SUN, 2002)

2.4 JAVA e J2SE

Java é uma linguagem de programação orientada a objetos de alto nível criada pela Sun *Microsystems* em 1991. Uma das características que mais se destaca nessa linguagem é a capacidade do código fonte ser compilado uma única vez e poder ser executado em qualquer arquitetura de *hardware* e *software* compatível com essa tecnologia. A Sun (SUN, 2008a) denomina essa característica como sendo *"Write Once Run Anywhere"*. Para atingir essa independência de plataformas, um programa Java necessita de uma máquina virtual instalada, que é chamada de JVM (Java *Virtual Machine*) . O código fonte é traduzido em *bytecodes* que a JVM consegue ler e interpretar. Breves (BREVES; P., 2007) define *bytecode* como: "Toda interface, classe, anotação e enumeração, além do respectivo código dentro deles, está em um arquivo .class, que tem um formato binário muito bem especificado, batizado de *bytecode*".

Java é uma tecnologia muito ampla, é possível programar desde dispositivos móveis a aplicações Web. Dentre as tecnologias desta linguagem, pode-se destacar Java EE (*Enterprise Edition*), SE (*Standard Edition*) e ME (*Micro Edition*). A Figura 2.4 apresenta um diagrama que mostra a abrangencia das diversas tecnologias Java.

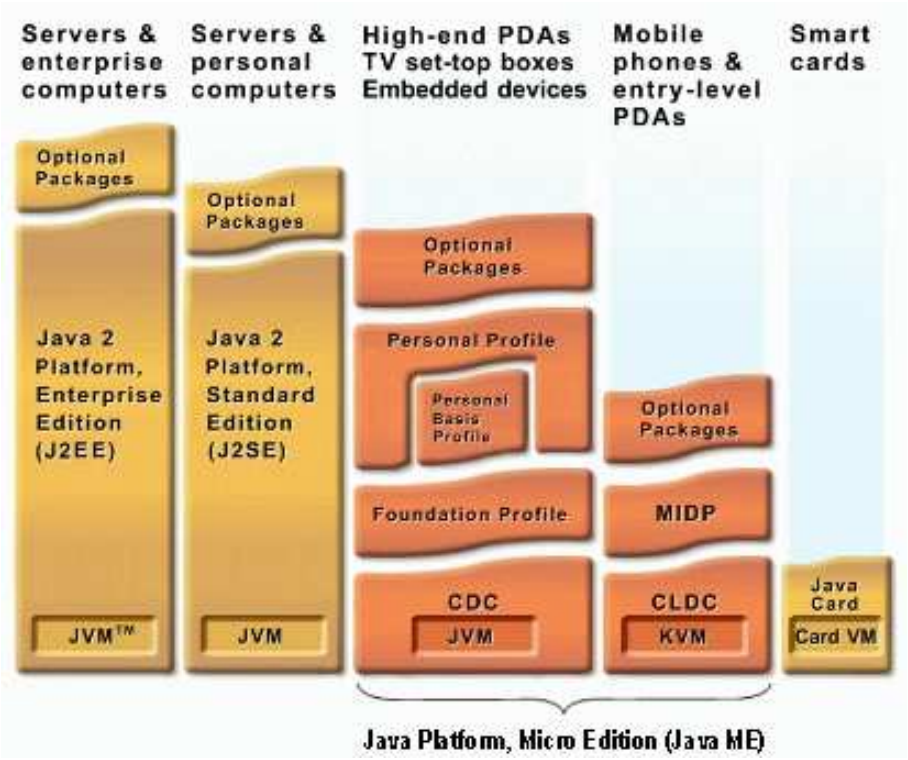

Figura 2.4: Tecnologias Java.(Fonte SUN, 2009b)

Tanto a plataforma J2SE (Java 2 *Platform, Standard Edition*) quanto a JSE (Java *Stan-*

dard Edition) tem o foco em sistemas *desktop*. A primeira refere-se às tecnologias Java da versão 1.2 até a versão 5. Já a nomenclatura JSE foi adotada a partir da versão 6 da linguagem Java. Sabe-se que atualmente a versão 5 é suportada pela maioria das plataformas computacionais. É o caso do sistema operacional MacOSX Tiger, que só tem suporte à versão 5 dessa linguagem.

3 O *FRAMEWORK* PAROLA

O Parola é um *framework* que encapsula uma série de decisões de projeto para a criação de aplicações de processamento e análise de imagens baseadas em *Plug-ins*. Ele implementa uma arquitetura plugável, facilmente extensível que controla os *plug-ins* e gerencia as imagens. Para isso, é necessário um sistema bem organizado, que se fundamenta em conceitos de engenharia de *software* e padrões de projeto para apresentar um *design* consistente. Do ponto de vista dos *plug-ins*, pode-se destacar que eles apresentam uma grande independência do restante do sistema, sendo limitados somente pelas interfaces dos objetos de fronteira. Esse capítulo tem como objetivo mostrar os principais aspectos da arquitetura e da tecnologia empregadas na construção do Parola, além de apresentar a metodologia de construção de um *plug-in*. Evita-se entrar em aspectos mais específicos da linguagem de programação, contudo algumas exceções se fazem necessárias para um melhor entendimento do trabalho.

3.1 Considerações de Tecnologia

Java é uma linguagem de programação de alto nível bastante conceituada na indústria de *software*. Diversas características a tornam a escolha preferida por arquitetos de *software* para os mais diversos projetos. Ao tratar, em específico, do *framework* Parola, podem-se destacar as seguintes características:

1. Orientação a objetos;
2. Invocação de objetos e métodos em tempo de execução (desconhecidos durante a compilação);
3. Existência de uma grande comunidade de desenvolvedores;

4. Existência da API (*Application Programming Interface*) de processamento de imagens JAI (*Java Advanced Imaging*);

5. Invocação de código desenvolvido em outras linguagens através do *framework* JNI (*Java Native Interface*);

6. Existência de bons *frameworks* para a construção de interfaces gráficas, em específico o *framework Swing* que faz parte da API padrão da plataforma J2SE/JSE;

7. Bom suporte a manipulação de arquivos XML (Extensible Markup Language).

A orientação a objetos já se consagrou como uma abordagem de desenvolvimento de sucesso. Já a invocação de métodos e objetos em tempo de execução não é uma característica que todas as linguagens orientadas a objetos possuem. Em Java, isso é possível por meio da API *Reflection*. A Sun (SUN, 2008b) afirma que a tecnologia *Reflection* possibilita que programadores construam programas que modifiquem o seu comportamento em tempo de execução, o que é necessário para a invocação dos *plug-ins*.

A API JAI é outro ponto importante, pois oferece um conjunto de interfaces orientadas a objetos que dão suporte a um modelo de programação de alto nível, para a manipulação de imagens digitais (SUN, 2009c). A JAI ainda permite que qualquer algoritmo de processamento de imagens possa ser adicionado à sua API e ser usado como se fosse uma parte nativa da biblioteca. Sendo assim, conforme a conceituação de *framework*, pode-se inferir que a tecnologia JAI é um *framework* que atua no domínio matemático de processamento e análise de imagens.

JNI, por usa vez, é uma tecnologia que se destaca, possibilita ao desenvolvedor tirar proveito da plataforma Java e, ainda, utilizar código escrito em outras linguagens (LIANG, 1999). Apesar do *framework* Parola não se beneficiar diretamente da tecnologia JNI, essa característica é considerada importante, pois permite que *plug-ins* se utilizem de tecnologias de processamento de alto desempenho em GPU (*Graphics Processing Unit*), como a tecnologia CUDA (*Compute Unified Device Architecture*) da Nvidia (CUDA é uma arquitetura para processamento paralelo de propósito geral disponível para a linguagem C (NVIDIA, 2008)).

Segundo a W3C (W3C, 2009) XML é um formato de arquivo texto simples e flexível que organiza as informações de maneira estruturada e clara através do uso de *tags*, de maneira similar ao formato HTML (*HyperText Markup Language*). XML é importante

por ser um padrão utilizado pela indústria de *software*. Diversas aplicações fazem uso desse formato de arquivo, utilizam-no, em especial, para guardar configurações e servindo de base para diversos arquivos dados. Além disso, ele é utilizado em diversos protocolos de comunicação, como, por exemplo, o protocolo SOAP (*Simple Object Access Protocol*). Suas características de simplicidade e flexibilidade permite a criação de arquivos de fácil entendimento. O grande grau de importância do XML, aliado ao ótimo suporte para à linguagem Java, foi fundamental para a definição desse formato para os arquivos de configuração do *framework* Parola.

3.2 Descrição funcional do Parola

Abaixo, segue a lista das principais funcionalidades que o Parola implementa:

1. Gerenciamento de *plug-ins*: Cada *plug-in* possui um controlador que gerencia o estado habilitado ou desabilitado;
2. *Plug-in* sempre habilitado (caso configurado);
3. Personalização da interface gráfica através de pacotes de estilo (conhecidos em inglês como *look and feel*);
4. Separadores entre os *plug-ins*;
5. Construção de menus e submenus (ordenados conforme o arquivo de configuração);
6. Construção da barra de tarefas;
7. Componente de visualização de imagens;
8. Componente de abertura de imagens;
9. Componente para salvar imagens;
10. Configuração da funcionalidade "sobre"(*tag about*, que gera um *pop-up*, utilizado normalmente para créditos);
11. Comando para organizar as janelas internas (cascata ou lado a lado).

Em relação ao objetivo de gerenciamento de *plug-ins* a Figura 3.1 apresenta duas imagens de uma aplicação construída com o Parola, colocadas lado a lado. No caso

A, evidencia-se um *plug-in* que só trabalha com imagens em tons de cinza, que é o *CalcPhase*. Já no caso B, visualiza-se um outro *plug-in* que só trabalha com imagens coloridas, o *ColorSpace3D*. Contudo, ambas imagens apresentam um terceiro e quarto *plug-ins*, que estão disponíveis tanto para imagens coloridas, quanto para imagens em tons de cinza, que são o de correção de iluminação e o de histograma. Essa funcionalidade, de gerenciar os estados habilitado e desabilitado dos *plug-ins*, é de extrema importância para o Parola, pois evita que os usuários finais do sistema executem um *plug-in* que receba uma imagem que não tenha sido projetado para trabalhar. Nesse caso, o problema se dá pois um *plug-in* pode gerar dados inconsistentes ou apresentar funcionamento anômalo apartir de uma imagem errada, gerando instabilidade para todo o sistema.

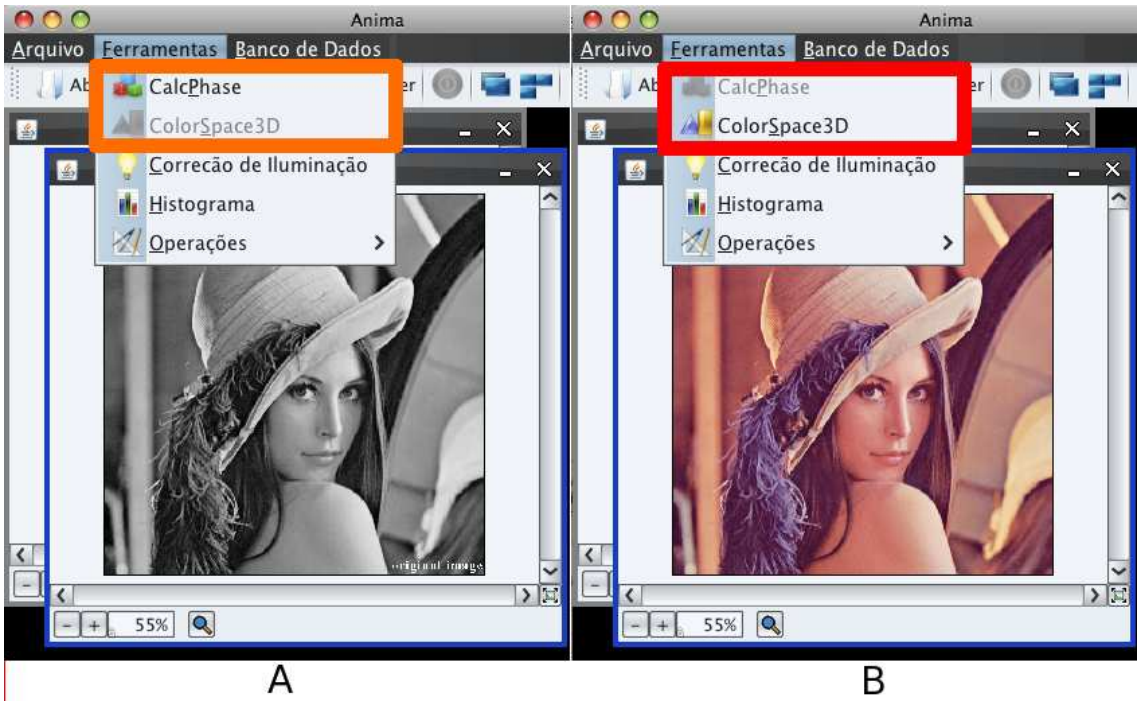

Figura 3.1: *Plug-ins* habilitados ou desabilitados conforme o tipo da imagem em foco.

3.3 Arquitetura e implementação

Bezerra (BEZERRA, 2007) afirma que em Sistemas de Software Orientados a Objetos "os objetos encapsulam tanto dados quanto comportamento. O comportamento de um objeto é definido de tal forma que ele possa cumprir com suas responsabilidades. Uma responsabilidade de um objeto é uma obrigação que este tem no sistema no qual ele está inserido. Graças às suas responsabilidades, um objeto colabora com outros objetos para que os objetivos do sistema sejam alcançados".

Tendo por base que um objeto encapsula tanto dado quanto comportamento dentro da categorização de objetos BCE (*Boundary, Control* e *Entity*), podem-se distinguir três categorias: os objetos de fronteira, de controle e de entidade. Bezerra (BEZERRA, 2007) os define da seguinte maneira:

1. Objetos de entidade normalmente servem como um repositório para alguma informação manipulada pelo sistema;

2. Os objetos de controle, chamados também controladores, são responsáveis por coordenar a execução de alguma funcionalidade específica do sistema. Esses objetos decidem o que o sistema deve fazer quando ocorre um evento relevante, ou seja, servem como “gerentes” de outros objetos para a realização de um ou mais casos de uso;

3. Os objetos de fronteira permitem o sistema interagir com o ambiente externo, tanto na comunicação com atores, quanto na interface com sistemas externos.

Além desses mencionados, pode-se afirmar, ainda, que existem objetos que são responsáveis pela criação de outros objetos, tratam-se de objetos de criação. A terminologia objetos de criação é baseada a partir da divisão que Gamma (GAMMA et al., 2005) propõem aos padrões de projeto e que defini três categorias: os padrões de criação, os padrões estruturais e os padrões comportamentais. Objetos de criação, normalmente, são classificados pela BCE com objetos de controle, devido as suas responsabilidades. Contudo, com o objetivo de facilitar o entendimento da arquitetura do Parola e pelo grau de especificidade desses objetos na construção deste *framework* tratar-se-ão separadamente.

Somando-se a isso, é possível afirmar que um objeto pode ser enquadrado em mais de uma categoria, mediante a suas responsabilidades. Um objeto, então, pode assumir a característica de controlador e de fronteira, como no caso do objeto de nome “Parola” (mesmo nome de sua classe). Esses tipos de objetos são tratados, logo abaixo, levando em conta a forma que o Parola foi implementado. A Figura 3.2 apresenta uma versão simplificada do diagrama de classes do *framework* Parola. As classes do pacote *model* foram subtraídas desse diagrama para garantir uma melhor compreensão, contudo, é possível ver em detalhes as classes pertencentes ao pacote *model* na Figura 3.3.

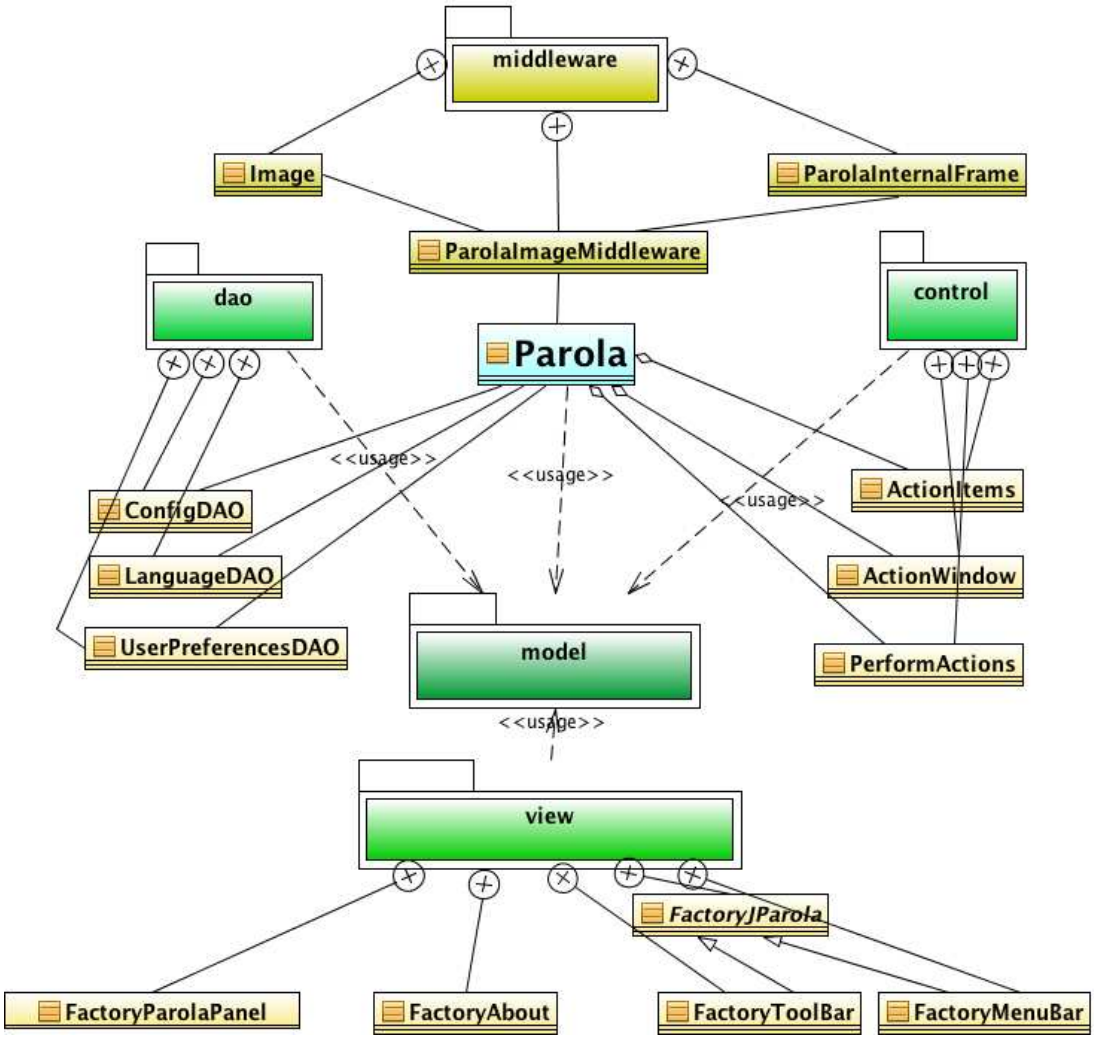

Figura 3.2: Diagrama De Classes do Parola - Simplificado.

3.3.1 Objetos de Entidade

O Parola, por tratar-se de uma ferramenta voltada para a construção de outras aplicações, apresenta um domínio de negócio - entidades - peculiar. Para projetistas de *software*, ver a relação entre as entidades e as tabelas de um banco de dados em um sistema de informação convencional, é uma tarefa bastante corriqueira. Mapeiam-se os relacionamentos entre as tabelas, assim, gerando as entidades. O diagrama de classes do domínio é relativamente de fácil entendimento tanto pelos desenvolvedores, quanto pelos usuários finais. A mesma coisa acontece com o Parola. Seus usuários, os desenvolvedores de *plug-ins*, conseguem facilmente identificar elementos de interfaces gráficas com o usuário.

Componentes comumente vistos em interfaces gráficas são representados e armazenados em instâncias das classes do diagrama da Figura 3.3. Pode-se destacar, nessa figura, que são representados menus, barra de tarefas (*ToolBars*), *Items* (forma pela qual são

inseridos *plug-ins* no Parola), entre outros.

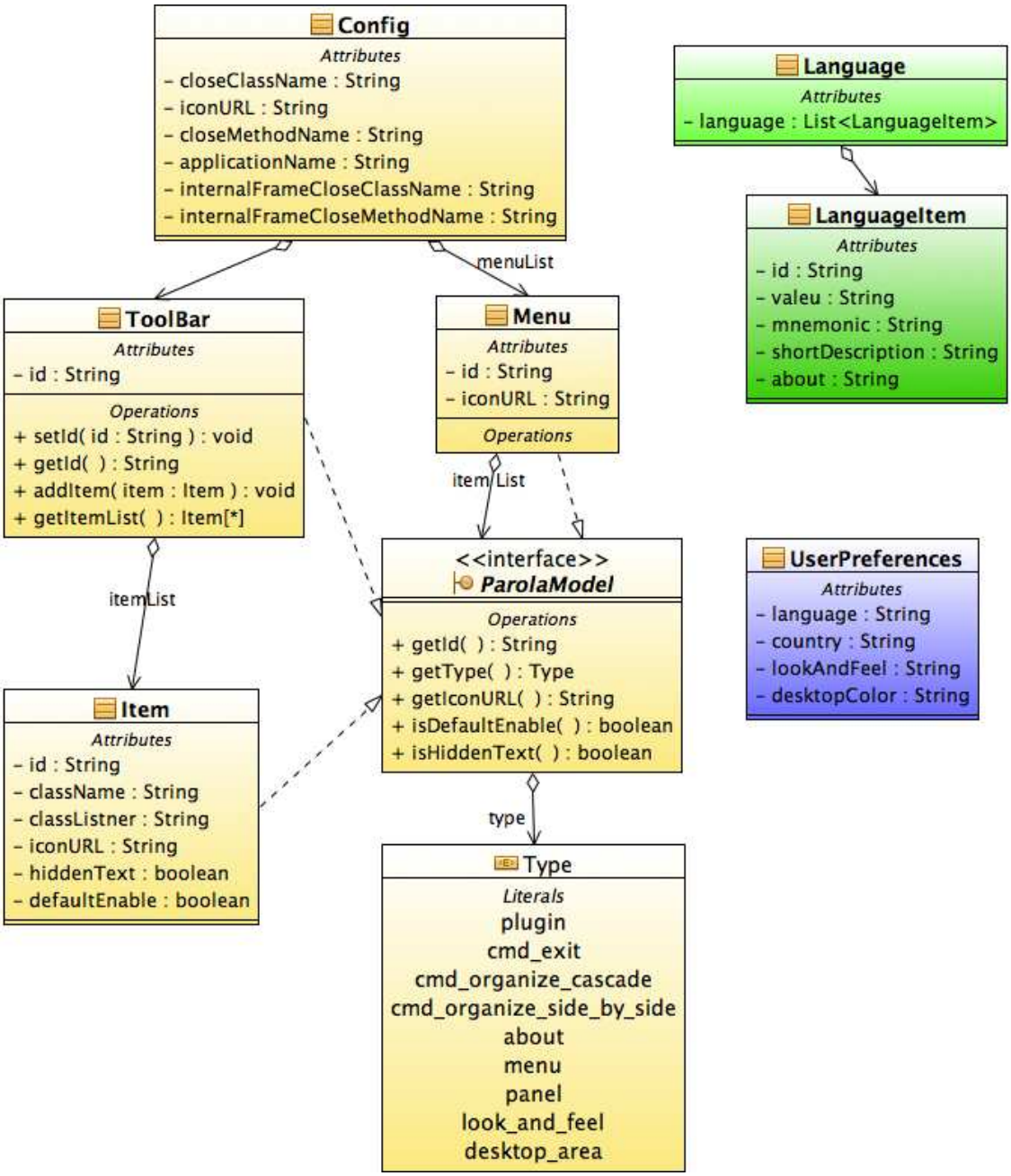

Figura 3.3: Diagrama de Classe de Domínio (Pacote *Model*).

Aprofundando a análise em relação a classe *Item*, é possível identificá-la como sendo de fundamental importância para o Parola. Isso se deve ao fato dela armazenar informações de configuração para *plug-ins*, como a classe a ser invocada (o *plug-in* propriamente dito) e a classe responsável pelo controle do *plug-in* (habilitando ou não), além de definir se o *plug-in* será habilitado por padrão (*defaultEnable*), entre outras configurações. Além da configuração dos *plug-ins*, essa classe assume outras responsabilidades que são herdadas da interface *ParolaModel*, destacando-se as listadas abaixo:

1. Definição de separadores entre os *plug-ins*;
2. Definição da função *about*;
3. Comandos do sistema:
   (a) *cmd_exit*: Dispara evento para fechar o programa;
   (b) *cmd_organize_cascade*: Dispõe as janelas internas em forma de cascata;
   (c) *cmd_organize_side_by_side*: Dispõe as janelas internas lado a lado;

A Figura 3.3 apresenta três tipos de classes do modelo. As classes que estão em tons de amarelo são responsáveis pelas informações de interface gráfica; as classes em tons de verde representam os dados de linguagem; e, por fim, a classe em tom de roxo armazena as preferências do usuário.

Os objetos de entidade são carregados a partir de três arquivos XML:

1. *config.xml*: Entidades de interface gráfica;
2. *main_<lingua>_<país>.xml*: Entidades de linguagem. Nesse caso, existe um arquivo para cada idioma. Por exemplo: *main_pt_BR.xml* (arquivo para português brasileiro) ou *main_en_US.xml* (inglês americano);
3. *userPreferences.xml*: Entidade de preferência do usuário. (Guarda as informações como língua, país e aparência).

3.3.2 Objetos de Criação

No Parola, as classes que apresentam o sufixo “DAO” e o prefixo “*Factory*” (fábricas) são responsáveis pela criação de objetos. O primeiro caso, corresponde ao uso do padrão de projeto *Data Access Object*, padrão que auxilia na interação com uma base de dados, para o acesso aos arquivos XML, gerando os objetos de entidade. No segundo caso, temos classes com o prefixo *Factory*, que são responsáveis pela criação dos componentes visuais. A partir dos objetos de entidade, as fábricas criam objetos de interface gráfica com as características necessárias a cada função. A Figura 3.2 apresenta tanto classes DAO, quanto *Factory*.

3.3.3 Objetos de Fronteira

A comunicação entre *Plug-ins* e o ambiente da aplicação é fundamental, e se dá através de duas classes principais, que são instanciadas gerando os objetos de fronteira. O primeiro caso, refere-se ao objeto *Parola*, que é a única instância da classe principal do *framework* e disponibiliza as interfaces básicas de acesso ao ambiente da aplicação. Pode-se destacar que esse objeto é mais útil para gerenciar informações de propósito geral, como:

1. Língua e o país que o ambiente está usando, sendo útil para que os *plug-ins* apresentem as suas interfaces gráficas na língua correta;
2. Informações para posicionar o *plug-in* em relação à janela da aplicação;
3. Interfaces de acesso de baixo nível, usadas quando são necessárias interações mais específicas.

Outro objeto responsável pela comunicação com os *plug-ins* é o *ParolaImageMiddleware*. Esse implementa uma camada de abstração para interagir com as imagens do sistema, adicionando novas e gerenciando imagens abertas. Implementa mecanismos que buscam imagens por tipos. Por exemplo, é possível ter acesso a todas as imagens em escala de cinza através do método *getAllGrayScaleImagesFrames()*. A Figura 3.4 mostra o diagrama de classes dos objetos de fronteira. Observa-se nesse diagrama os métodos que servem de interface de comunicação com os *plug-ins*. Questões mais aprofundadas sobre esse diagrama entrariam no domínio específico da linguagem, o que foge ao escopo desse trabalho.

Com o objetivo de facilitar o entendimento das comunicações dos objetos de fronteira com os *plug-ins*, vê-se na Figura 3.5 as possíveis combinações. Dessa forma, os *plug-ins* podem se comunicar diretamente com o Parola, usando o *ParolaImageMiddleware* ou através das duas classes. Aprofundando a análise sobre essa figura pode-se observar as seguintes características:

- O *plug-in* 1 comunica-se diretamente com o objeto *Parola* – esse caso acontece quando existe alguma funcionalidade que não manipula imagens, por exemplo, um *plug-in* para visualização de dados gerados por outro *plug-in*;

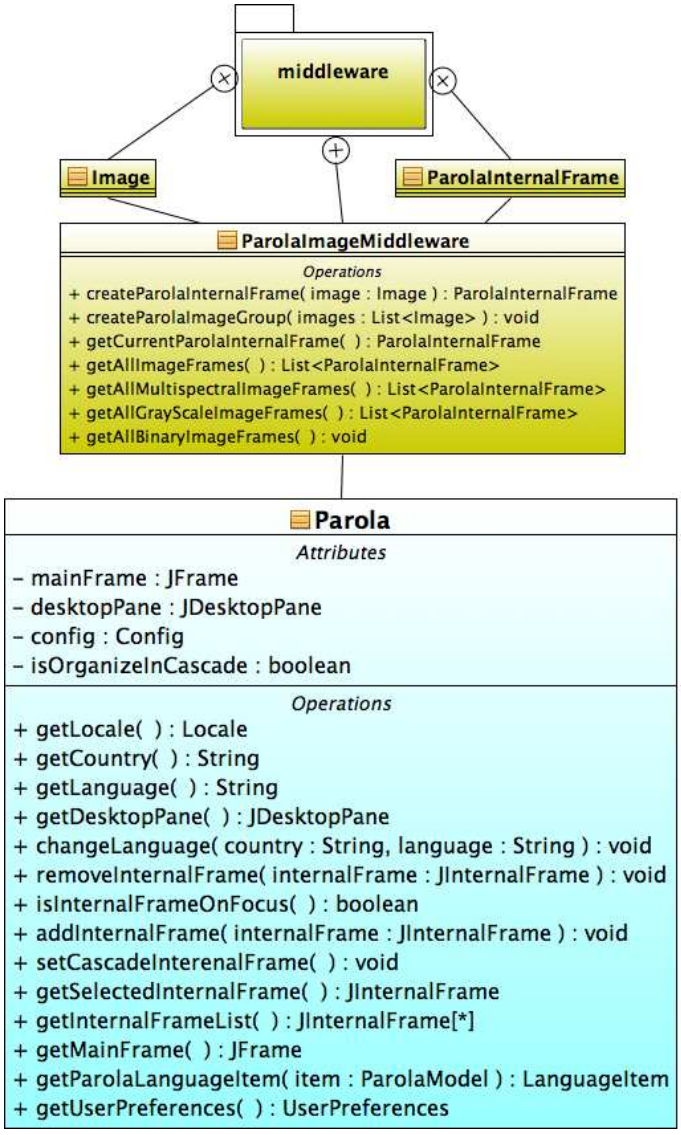

Figura 3.4: Diagrama de Classe - Classes que se comunicam diretamente com os *plug-ins*.

- O *plug-in* 2 faz uso da comunicação com os objetos *Parola* e *ParolaImageMiddleware* – esse caso concentra-se na grande maioria dos *plug-ins*. Certas funcionalidades que interagem com o usuário necessitam saber, no mínimo, dados da língua e a informações para posicionar a janela do *plug-in*, além de ter que interagir com o sistema para buscar, atualizar ou criar novas de imagens;

- O *plug-in* 3 comunica-se somente com o *ParolaImageMiddleware* – essa situação acontece quando uma funcionalidade não precisa de interface gráfica. Por exemplo, quando se converte uma uma imagem RGB para escala de cinza ou quando se separam as suas bandas.

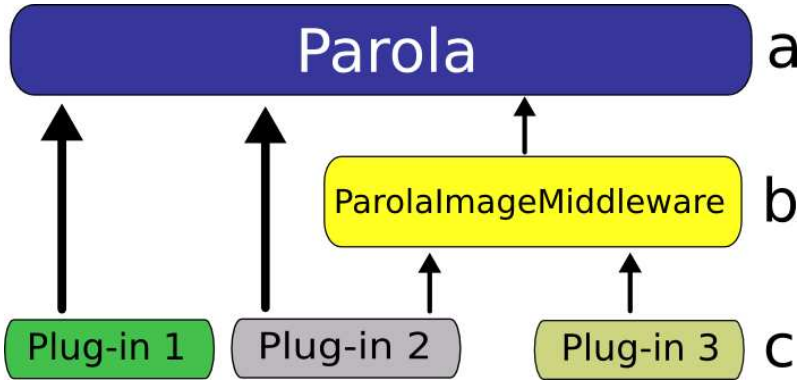

Figura 3.5: Camadas de comunicação - objetos de fronteira e *plug-ins*.

3.3.4 Objetos de Controle

Desenvolver *plug-ins*, gerenciar imagens, ler arquivos de configuração e construir interface gráfica com o usuário são tarefas inerentes ao propósito do Parola. Mas faz-se necessário invocar a classe certa na hora certa, habilitar ou não *plug-ins* e mesmo garantir a construção correta da interface gráfica. Essas ações, referem-se aos objetos de controle. No Parola dividem-se em dois grupos: Os controladores dos *plug-ins* e os controladores do Parola.

Os controladores dos *plug-ins* reagem a eventos gerados pelo Parola, respondendo, habilitando ou desabilitando os *plug-ins*, dependendo do estado da aplicação. Cada *plug-in* pode ter somente um controlador associado ou não possuir nenhum. No caso de não se ter nenhum controlador associado o *plug-in* se manterá habilitado ou não, mediante a sua configuração *defaultEnabled*. Pode-se verificar os diversos estados que a aplicação pode assumir no diagrama de estados da Figura 3.7. Assim sendo, os controladores dos *plug-ins* representam um caso de uso típico do padrão *observer*, visto que esses controladores ficam "observando" o estado da aplicação. Em Java utiliza-se do termo *listner* para se referir ao observador. Esse termo, por sua vez, é bastante usado pelo *framework* Swing. Dessa forma, os controladores são definidos no arquivo de configuração pela *tag classListener*. Outro ponto importante é que seu funcionamento é bastante simples, para isso o controlador possui apenas um método que retorna *true*, no caso habilitado, ou *false*, no caso desabilitado.

Os controles de estados do Parola são os mecanismos responsáveis por informar o novo estado para os controladores dos *plug-ins*, além de invocar a classe certa na hora em que a aplicação será finalizada. A Figura 3.2, que mostra o diagrama de classes do Parola,

possibilita a observação do pacote *controler*, que apresenta, por usa vez, três classes:

- *PerformAction* – Trata os eventos gerados pela troca, perda ou ganho do foco sobre as imagens no sistema. A cada evento o Parola avisa seu estado aos observadores e cada observador, de posse do novo estado da aplicação, retorna um valor *booleano* para o *framework* o avisando se o *plug-in* de sua responsabilidade está ou não habilitado;

- *ActionItems* – A partir do momento em que o *plugin* encontra-se habilitado, essa classe é responsável pela sua execução. Em outras palavas, ela cria uma instância do *plug-in*;

- *ActionWindow* – Invoca a classe responsável pela ação de fechamento da aplicação.

3.3.5 Inicialização do Parola

Esta seção trata dos passos que são necessários para o Parola ser inicializado. O sistema lê os arquivos de configuração e inicializa todos os objetos na memória, assim, a aplicação está pronta para uso. Objetos de entidade, criação, fronteira e controle são inicializados e começam a atuar juntos para criar um ambiente completo. Em especial serão tratado os pontos que envolvem a criação dos objetos de entidade até a construção da interface gráfica com o usuário.

A Figura 3.6 apresenta o diagrama de sequência dos objetos de criação. Primeiramente, o objeto *Parola* - objeto principal - faz a requisição dos elementos de configuração ao *ConfigDAO*, e recebe, como resposta, os objetos de entidade, carregados a partir do aquivo *config.xml*. Esse arquivo de configuração contem dados sobre os *plug-ins*, além de diversas outras informações como, por exemplo, a ordem dos *plug-ins* e de mais elementos na interface gráfica. Em um segundo momento, o objeto principal do *framework* recebe os dados de preferência de usuário. A partir dessas informações, é notificado o objeto *LanguageDAO* a língua e o país, dados necessários para a tradução correta do sistema. De posse dos objetos de entidade, o Parola repassa as informações necessárias para os objetos com o prefixo *Factory*, a fim de que sejam criados os objetos de interface gráfica (janela da aplicação, menus, submenus, barra de tarefas, entre outros). E por fim, para cada elemento da interface gráfica existe um *ParolaModel* associado, que permite ao DAO de linguagem buscar os dados para o Parola fazer a tradução do sistema. A partir desse momento, o sistema encontra-se inicializado, esperando a interação com o usuário.

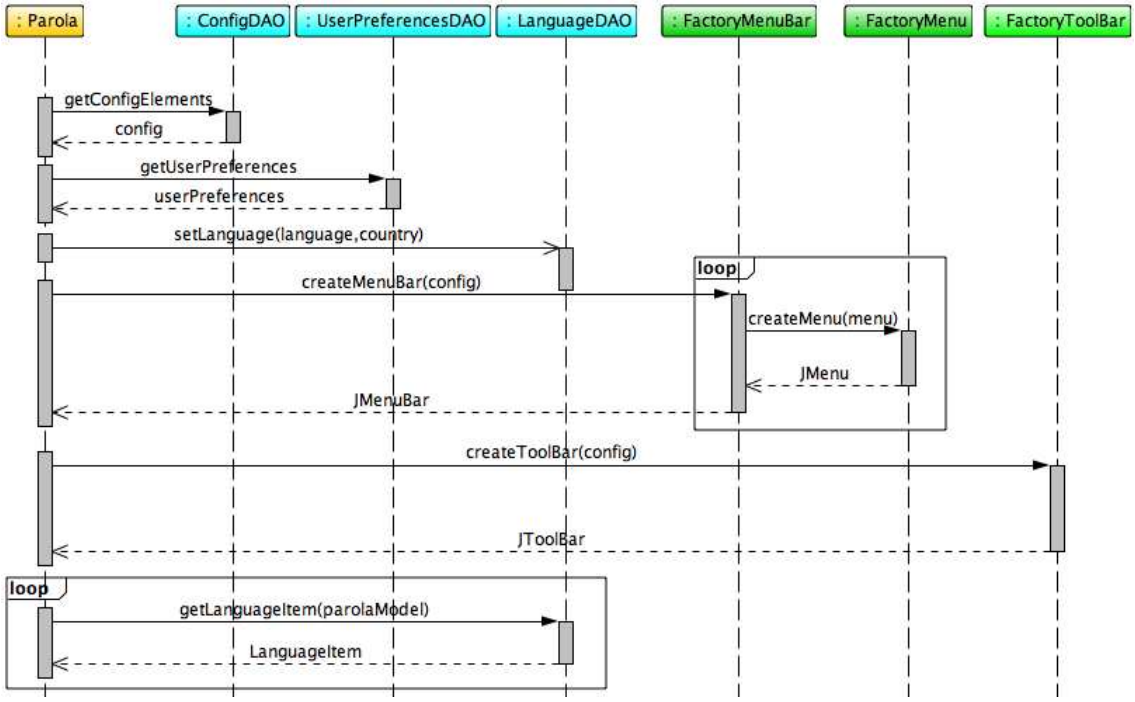

Figura 3.6: Diagrama de sequência: Inicialização do Parola.

3.4 Internacionalização

A internacionalização é um aspecto importante para todas as aplicações que buscam ter uma comunidade global de usuários. O Parola apresenta seu idioma baseado em arquivos XML, uma vez que ele necessita conhecer uma série de informações para a criação da interface gráfica como, por exemplo, mnemonicos (atalhos de teclado) e descrições (usadas quando se repousa o mouse sobre algum *item*). O *framework* necessita de objetos que encapsulem esses dados de linguagem para que se possa ter uma implementação clara e simples, destacam-se os objetos de entidade responsáveis pela língua, que assumem essa função.

Abaixo, é apresentado o exemplo do código de um arquivo simplificado de idioma:

```
<language>
    <item>
        <id>meu_plugin_id</id>
        <value>Isso vai aparecer no botão do plugin</value>
        <mnemonic>I</mnemonic>
        <shortDescription>
Quando o mouse ficar parado sobre o plugin,
```

```
essa é a mensagem que vai aparecer em uma caixinha
amarela.
        </shortDescription>
    </item>
</language>
```

Contudo, a forma de internacionalização baseada em XML pode não ser a melhor saída para todos os casos. Os *plug-ins* de posse de informações de localidade (país e língua), que facilmente conseguem obter a partir do objeto de fronteira *Parola*, podem implementar a sua tradução da maneira que lhe for mais conveniente. Entretanto, é importante destacar a importância de usar formas padronizadas para implementar a internacionalização de sistemas de *software*. A padronização facilita a tradução do sistema para um determinado idioma, pois torna mais simples o trabalho dos responsáveis pela tradução, os tradutores, que tendo que se preocupar com diversas formas de internacionalização gastam mais tempo e, consequentemente, elevam os custos do processo.

Dessa forma, é encorajado o uso de arquivos "*.properties*", padronização definida pela Sun *Microsystem*. De uma maneira resumida, existe um arquivo de texto com a terminação "*.properties*" para cada país e língua a ser traduzida, para cada *plug-in*. Todos os arquivos de linguagem apresentam uma coluna que representa a chave e uma segunda que representar a sua tradução, separadas pelo caractere '=' (igual). Através de mecanismos que a API padrão Java implementa facilmente se tem acesso as frases que serão usadas para a tradução do *plug-in*.

Segue-se o exemplo de um arquivo "*.properties*" em português brasileiro. Esse arquivo possuirá o nome: "*plug-in_pt_BR.properties*".

```
PalavraChave1 = Essa é a tradução da PalavarChave1.
```

Abaixo, o código de um arquivo "*.properties*" em inglês americano, e seu nome é "*plug-in_en_US.properties*".

```
PalavraChave1 = This is the  PalavarChave1's translation.
```

3.5 Criação e instalação de *plug-ins*

A inserção de novas funcionalidades ao *framework* se dá através de sua parte flexível ou *hotspots*, conhecidas como *plug-ins*. Para a elaboração de um novo *plug-in* deve-se

criar uma classe que possua um construtor que não receba argumentos. Isso, é necessário para a instanciação do *plug-in* em tempo de execução. As regras básicas de codificação para a criação de um *plug-in* são:

1. Criação de um construtor que não recebe argumentos, buscando seguir o padrão JavaBeans (SUN, 2009a);
2. Definir se o *plug-in* será ou não modal, ou seja, se deve bloquear as demais janelas do programa até o fim de sua execução ou não;
3. Definir posição em relação à janela principal;
4. E, finalmente, tornar o *plug-in* visível.

Abaixo, é apresentado o exemplo do código do construtor de um *plug-in*:

```
public SamplePlugin() {
   super();
   // inicializa os componentes do plug-in
   initComponents();
   // bloquear as demais janelas
   this.setModal(true);
   /*posiciona o plug-in centralizado em
    * relação a janela princial
    */
   this.setLocationRelativeTo(Parola.getMainFrame());
    //torna o plug-in visível
   this.setVisible(true);
}
```

Após a criação do construtor, é necessário configurar o Parola para executá-lo. Observa-se no próximo código a configuração de um *plug-in* (no arquivo *"config.xml"*). A estrutura que representa um *plug-in* é definida pela *tag "item"*, a *tag "id"*, por usa vez, refere-se ao identificador do *plug-in*, que será posteriormente utilizada para a tradução. O tipo *"plugin"* , na *tag type*, informa ao Parola que esse item se refere a um *plug-in*, e portanto, este deverá identificar as demais *tags* que o seguem. Já a *tag className* faz referência ao

endereço do *plug-in*. Por sua vez, o *classListner* é a classe que será invocada pelo Parola a fim de observar os eventos que possam habilitar ou não o *plug-in*, como, por exemplo, uma imagem ganhar o foco. No caso de não existir imagem em foco, o *plug-in* assume o estado padrão que foi definido pela *tag defaultEnable* (habilitado ou não). Pode-se visualizar a máquina de estados do Parola na Figura 3.7 e ter uma compreensão de como o Parola interage com os *plug-ins* e com os estados internos da aplicação.

```
 <item>
    <id>meu_plugin_id</id>
    <type>plugin</type>
    <className>
            br.com.animati.anima.SamplePlugin
    </className>
    <classListener>
            br.com.animati.anima.SamplePluginListener
    </classListener>
    <defaultEnable>false</defaultEnable>
</item>
```

Para o Parola poder ter acesso a esse novo *plug-in* pode-se proceder das seguintes maneiras:

1. De posse do novo *plug-in* e com os arquivos de configuração devidamente editados, faz-se necessário colocar o *plug-in* em um lugar visível ao *classpath*, que define os locais onde a JVM consegue ter acesso as classes Java. Isso se faz das seguintes maneiras:

   (a) Configurando o *classpath* através de um argumento de linha de comando, assim a JVM consegue encontrar as classes do *plug-in*. Isso se dá através do comando "java -classpath <meu_diretório_com_o_plugin> -jar <nome_do_projeto_feito_com_o_Parola>". No caso de um projeto chamado Parola far-se-ia da seguinte maneira: "java -classpath /diretório_com_o_plugin -jar Parola". Essa maneira de adicionar um *plug-in* é bastante útil pois permite ao programador que crie *scripts* os quais contenham esse comando, podendo

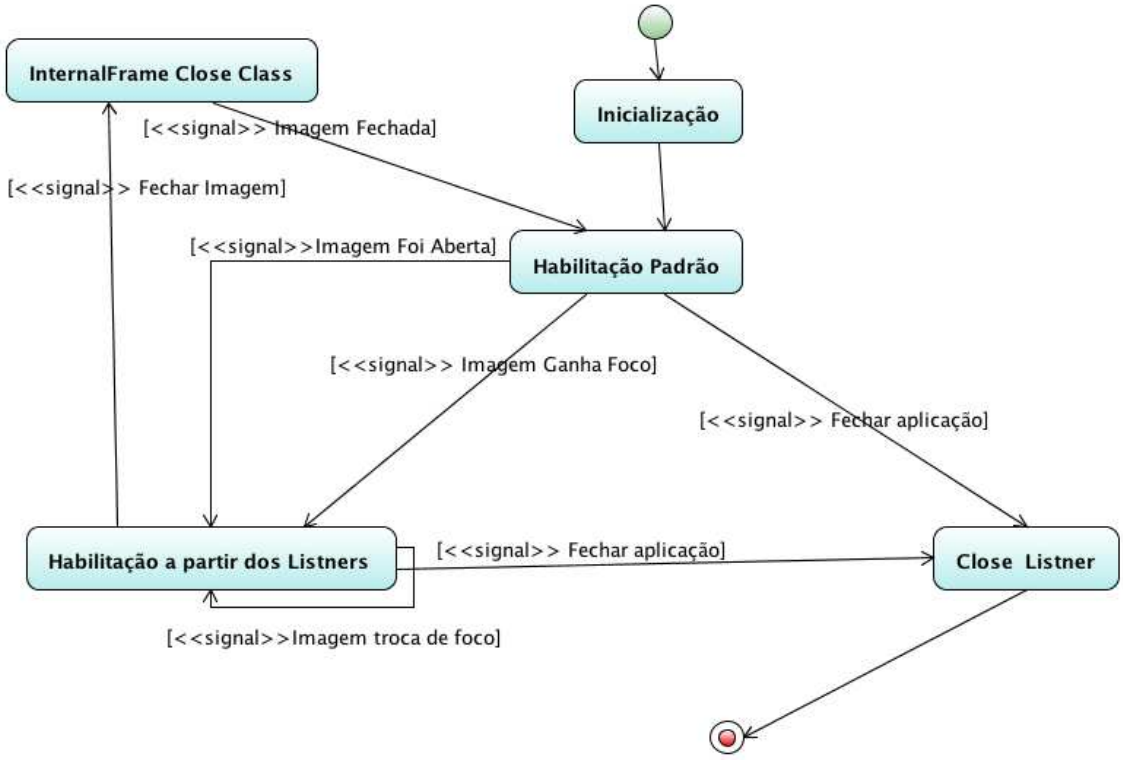

Figura 3.7: Máquina de estados do Parola - Eventos internos do Parola.

ser feito, por exemplo, através de arquivos ".bat"(em sistemas windows) ou ".sh"(em sistemas do tipo Unix).

(b) É possível, também, configurar a variável de ambiente do sistema operacional relativa ao *classpath*. Isso, contudo, varia dentre os sistemas operacionais. Quando configurada a variável de ambiente, para buscar as classes do *plug-in*, basta invocar a aplicação para ela pode executar o *plug-in*. Detalhes sobre essa configuração não serão abordados nesse trabalho por considerar a primeira alternativa a forma mais simples e fácil.

(c) A última maneira se faz por colocar as classes do *plug-in* em um diretório já mapeado pelo *classpath*. Considera-se essa alternativa útil quando o *plug-in* possa vir a ser útil a aplicações de terceiros. Contudo, não será abordada essa configuração por considerar a primeira alternativa a que se adapta melhor a maioria dos casos.

2. Outra forma de se adicionar funcionalidades, se faz a partir do código fonte do *framework*. Nesse caso, cria-se um projeto em uma IDE (*Integrated Development Environment*) como, por exemplo, o *NetBeans*. IDEs são ambientes de desenvol-

vimento de *software* que combinam editor, compilador, ferramenta para depuração de código entre outras funcionalidades. Cria-se então um projeto, adiciona-se os códigos do Parola e cria-se o pacote do *plug-in*.Quando compilado o projeto, normalmente, o *plug-in* será empacotado junto do pacote da aplicação. Dessa forma não se faz necessário explicitar o caminho do *plug-in* ao *classpath* da JVM.

Outro pronto importante é saber como se consegue ter acesso à imagem desejada e como se pode devolver uma imagem ao sistema. Nesse caso observa-se no código abaixo um exemplo de como o desenvolvedor faz para adquirir um objeto da classe *PlanarImage*, usada pela JAI para os métodos de processamento de imagens.

```
ParolaInternalFrame animaInternalFrame =
ParolaImageMiddleware.getCurrentParolaImageFrame();
PlanarImage planarImage =
animaInternalFrame.getImage().getPlanarImage();
```

Quando do termino do processamento, deve-se enviar a nova imagem ao Parola através do *ParolaImageMiddleware* conforme o código abaixo:

```
Image image = new Image();
image.setPlanarImage(newImage);
ParolaImageMiddleware.createParolaInternalFrame(image);
```

Essa seção apresentou os conceitos básicos para o desenvolvimento e instalação de novas funcionalidades em sistemas baseados no *framework* Parola.

4 RESULTADOS OBTIDOS

O *framework* descrito nas seções anteriores foi utilizado com sucesso por vários alunos de iniciação científica do Laboratório de Computação Aplicada da Universidade Federal de Santa Maria (LaCA/UFM). Além de alunos do LaCA/UFSM a empresa Animati Computação Aplicada teve alguns de seus desenvolvedores trabalhando no sistema Anima (Figura 4.1). O Anima é um sistema de processamento e análise de imagens desenvolvido através de uma parceria entre a Animati e o laboratório LaCA. Esse capitulo apresenta um comparativo da abordagem proposta pelo sistema Arthemis, antigo projeto do LaCA, em comparação ao Anima, desenvolvido com o Parola.

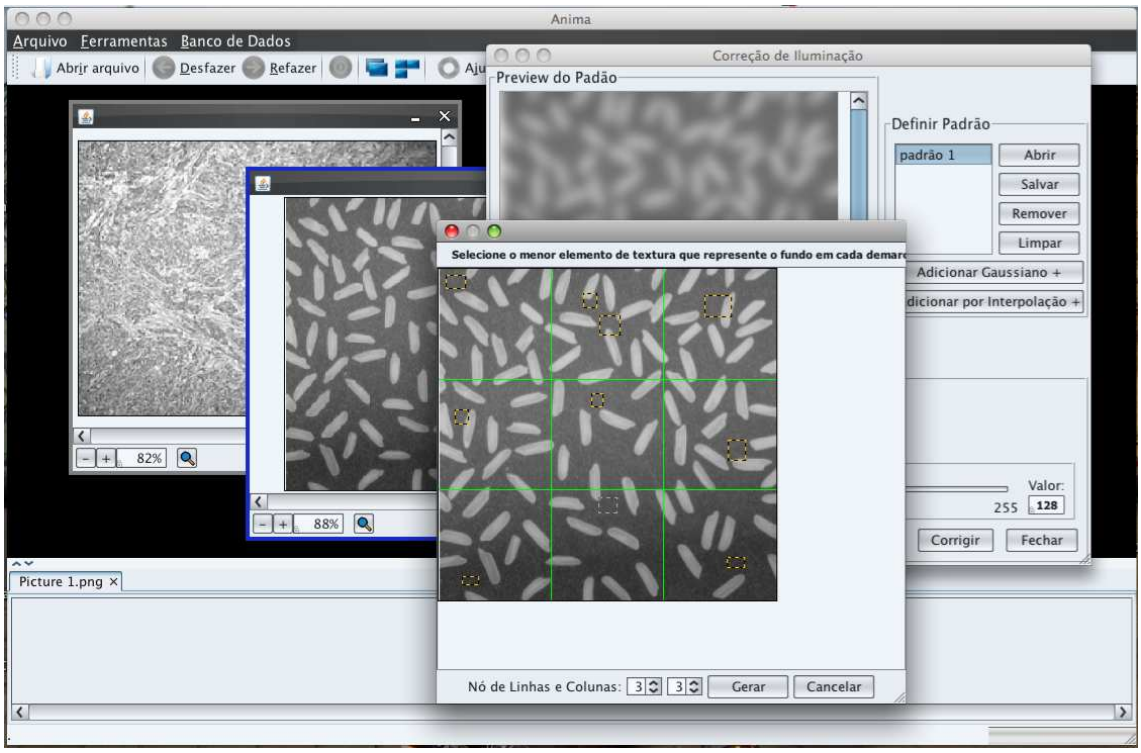

Figura 4.1: Anima - Aplicação de Processamento de Imagens (Animati, LaCA/UFSM) .

O Arthemis é uma aplicação de processamento e análise de imagem que apresenta a

mesma proposta do Anima, isto é, busca ser uma plataforma computacional extensível e personalizável para processar imagens digitais. Os dois *softwares* apresentam arquiteturas bem diferentes. O Arthemis segue a proposta de arquitetura três camadas, baseada em componentes, divididas da seguinte maneira: A primeira camada, representa a APJ JAI. A segunda, se refere aos métodos de processamento de imagens desenvolvidos no LaCA, esse métodos fazem acesso à primeira camada. E a terceira camada, é a camada de interface gráfica, que faz acesso aos métodos da segunda camada. O Anima, por outro lado, é uma coleção de *plug-ins* inseridos no Parola, tirando vantagem da proposta arquitetural disponibiliza pelo *framework*. Sendo assim, as características estruturais e arquiteturais atribuídas ao Anima na verdade se referem ao Parola, contudo será usado o nome Anima em alguns casos por representar uma aplicação voltada ao usuário final.

4.1 Instalação de novas funcionalidades (Arthemis X Anima)

O Arthemis configura a interface gráfica através de arquivos XML, de maneira similar aos recursos do Parola. Entretanto, sua configuração é bastate trabalhosa , visto que é necessário editar o código fonte de algumas classes Java além de ter que alterar o arquivo XML de configuração. Em outras palavras, para adicionar uma nova funcionalidade é necessário fazer o mapeamento da classe a ser invocado em pelo menos três classes Java, para daí usar as configurações dos arquivos XML. Desse modo, seus arquivos XML apenas se responsabilizam pela organização das funcionalidades na interface gráfica da aplicação.

Já o Anima, por ser baseado no Parola, necessita apenas a configuração do arquivo *config.xml* e dos arquivos de linguagens (para a internacionalização). Dispensa, assim, a necessidade de alteração do código fonte da aplicação. O que evidencia a primeira grande mudança de paradigma entre o Anima e o Arthemis. O Anima tem suas funcionalidades adicionadas na forma de *plug-ins* e o arquivo *config.xml* guarda as informações relevantes para a instanciação das funcionalidades, evita-se a prática do *hard coding* (prática que altera diretamente o código fonte da aplicação para fazer mudanças em sua configuração). O Parola utiliza-se da API *Reflection* de forma mais eficiente que o Arthemis.

4.2 Controlando as funcionalidades (Arthemis X Anima)

Fazendo-se uma análise da arquitetura três camadas implementada pelo Arthemis é possível fazer as seguintes observações:

1. A segunda camada, responsável pelos métodos de processamento de imagens, apresenta um boa independência em relação à camada de interface gráfica;

2. A terceira camada, a de interface gráfica, apresenta uma grande importância no sistema, pois possui as seguintes responsabilidades:

   (a) Criar a interface gráfica de toda a aplicação;

   (b) Implementar a interface gráfica de cada método de processamento de imagens;

   (c) Tratar todos os possíveis erros do programa;

   (d) Gerenciar as imagens abertas no sistema.

A arquitetura do Arthemis foi capaz de criar uma segunda camada com forte independência da interface gráfica. Suas características de implementação e padronização de interfaces de acesso deixavam aberta a possibilidade de se criar uma linguagem de macros. Entretanto, a terceira camada, fortemente baseada em componentes, apresentava deficiências bastante sérias. Não existia um mecanismo que garantisse que uma determinada funcionalidade ficasse disponível somente quando o estado interno do sistema fosse favorável. Quando muito, a interface gráfica capturava as exceções geradas pelo uso de uma imagem incompatível a um método matemático e notificava o usuário do sistema com alguma mensagem de erro. Resultando que o sistema, em muitos casos, gerasse resultados inconsistentes. Outras vezes, o sistema disparava exceções, das classes da primeira ou segunda camada, que quando não tratadas geravam funcionamento anômalo a aplicação. A camada de interface gráfica do Arthemis representa um verdadeiro desafio. Era altamente complexa e apresentava um fluxo de controle de difícil entendimento.

Um ponto chave do Anima é o controle que se tem sobre cada funcionalidade. Como explicado no capítulo anterior, o Parola disponibiliza formas de gerenciar as funcionalidades, que são inseridas na forma de *plug-ins*. Existe um controlador associado a cada *plug-in*, esse é responsável por observar o *status* interno do sistema, habilitando ou desabilitando o *plug-in* conforme a necessidade. Esse controle consegue resolver o principal problema de interface gráfica do Arthemis, pois ele evita que o problema aconteça.

Quando se tinha um erro em um determinado método de processamento de imagens, e se necessitava descobrir se o problema é de interface gráfica ou de implementação matemática a arquitetura do Arthemis se mostrava bastante complexa. Pois, o Arthemis apresentava uma estrutura distribuída. Os métodos matemáticos ficavam em um pacote específico e as interfaces gráficas ficavam em outro. Isso gerava uma estrutura bastante grande, e que necessitava muito tempo de análise para o seu entendimento. Soma-se a isso, o fato de que a camada responsável pela interface gráfica era fortemente baseada em componentes, isso acarretou uma estrutura monolítica, o que, nesse caso, dificultava a compreensão do sistema. Dessa forma o processo de depuração ficava prejudicado.

Por outro lado, o Anima apresenta a sua estrutura encapsulada em *plug-ins*. Isso significa que, cada *plug-in* carrega em seu pacote a interface gráfica e a implementação do método matemático. Assim, o processo de depuração fica restrito ao pacote do *plug-in*, o que simplifica ao processo de detecção de erros.

4.3 Anima: Lista de *plug-ins*

O Anima apresenta os seguintes *plug-ins* implementados:

1. CalcPhase : “É uma ferramenta para determinar as frações de área dos componentes homogêneos de uma imagem usando o método da análise linear de histogramas”(FRIEDRICH, 2007);

2. Correção de Iluminação: Responsável por implementar métodos de processamento de imagens para correção do *vignetting*, problema bastante comum em laboratórios que trabalham com microscopia óptica (BERNI, 2007);

3. ColorSpace3D: Ferramenta que permite a visualização 3D de diferentes espaços de cores (DARONCO, 2006; MAZZANTI, 2008);

4. Histogramas: *Plug-in* que permite a visualização dos histogramas de imagens digitais.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou uma arquitetura flexível baseada em *plug-ins* e padrões de projetos, visando a construção de um *framework* para processamento e análise de imagens, denominado Parola. O Parola torna a inserção de novas funcionalidades a um sistema uma tarefa simples, pois trabalha com o conceito de *plug-ins*. O uso de tecnologias baseadas em *plug-ins* proporciona maior liberdade ao programador na elaboração de novos módulos, em razão do *plug-in* não possuir uma dependência rígida em relação a arquitetura do sistema a que pertence. Isso se deve ao fato da interação com a aplicação ser feita através dos objetos de fronteira.

Essa abordagem permite que os sistemas possam dispor de uma grande quantidade de funcionalidades, sendo que somente as necessárias na execução de um determinado procedimento são carregadas em tempo de execução. A invocação de *plug-ins* por demanda aumenta a eficiência de utilização de recursos computacionais, pois torna o *software* mais leve e direcionado.

Outro ponto significativo é que a arquitetura que o Parola oferece garante um alto grau de independência entre os *plug-ins*. Isso representa uma boa opção para locais onde exista uma grande rotatividade de pessoas como em laboratórios de pesquisa de universidades. É importante destacar que não se faz necessário, por parte do desenvolvedor do *plug-in*, ter um conhecimento aprofundado sobre a arquitetura do sistema, pois a inserção de um *plug-in* se dá através de regras simples. Dessa forma, o desenvolvedor otimiza seu tempo, utilizando-o no desenvolvimento dos métodos matemáticos, enquanto as questões estruturais ficam a cargo do Parola.

Supondo-se, ainda, que um *plug-in* foi mal implementado ou que esteja gerando resultados errôneos, esse não compromete toda a aplicação, pois caso seja necessário, o *plug-in* pode ser facilmente desabilitado ou substituído por outro mais atualizado. Somando-se

a isso, uma aplicação de processamento de imagens possui diversos pontos de controle, além de ser responsável por gerenciar diversas imagens. Dessa forma, é mais econômico reescrever um *plug-in* do que toda a aplicação.

Com a utilização de padrões de projeto aplicam-se um conjunto de melhores práticas para a solução de problemas comuns em projetos de *software*, favorecendo a busca por reusabilidade de códigos. À medida que o sistema cresce, os padrões auxiliam no seu entendimento e extensibilidade, que são características necessárias a um *framework*, que busca encapsular uma série de descrições importantes do projeto de *software*.

Em relação aos aspectos estruturais internos ao Parola, pode-se destacar que a implementação dos controladores dos *plug-ins* seria mais complexa se não fosse a utilização do padrão *observer*. O padrão DAO, por usa vez, permitiu uma implementação do acesso aos dados dos arquivos XML de maneira simples e objetiva.

Como trabalhos futuros, pretende-se implementar interfaces para o desenvolvimento dos *plug-ins*, visando a criação de uma forma padronizada que permita aos *plug-ins* se comunicarem. Diversas funcionalidades poderão tirar proveito dessa característica, como, por exemplo, a implementação de uma linguagem de macro. Outros trabalhos futuros referem-se a comparação da arquitetura que o Parola oferece em relação a outros *frameworks* para processamento de imagens, além de buscar comparativos com outras arquiteturas baseadas em *plug-ins*.

REFERÊNCIAS

BERNI, J. C. A. **Desenvolvimento e Implementação de Métodos de Correção de Iluminação para Imagens Digitais**. Santa Maria: Curso de Ciência da Computação. Universidade Federal de Santa Maria., 2007.

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2.ed..ed. [S.l.]: Elservier, 2007.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. **The Unified Modeling Language User Guide**. [S.l.]: Addison-Wesley, 1999.

BREVES, A.; P., S. Desvendando o Bytecode. **Revista Mundojava (ISSN 1679-3978) , Curitiba**, [S.l.], n.23, maio 2007.

COOPER, J. W. **The Design Patterns - Java Companion**. [S.l.]: Addison-Wesley, 1998.

DARONCO, L. C. **Interface 3D para representação da distribuição de cores de imagens digitais em diferentes espaços de cores**. Santa Maria: Curso de Ciência da Computação. Universidade Federal de Santa Maria., 2006.

DEITEL, H.; DEITEL, P. **Java Como programar**. Proto Alegre: Bookman, 2004.

FAYAD, M. E-Frame: a process-based object-oriented framework for e-commerce. **International Conference on Computing**, Las Vegas, Junho 2001.

FAYAD, M. E.; SCHMIDT, D. C.; JOHNSON, E. R. **Building application frameworks object-oriented foundations of framework design.** [S.l.]: John Wiley Sons, 1999.

FAYAD, M.; HAMU, D. **Object-Oriented Enterprise Frameworks,**. New York: Wiley & Sons, 1999.

FREITAS, D. G. **Athena**: um framework para a construção de interfaces humano-computador no domínio da segmentação de imagens. 2006. Dissertação (Mestrado) — PPGEP-UFSM, Santa Maria.

FRIEDRICH, A. R. **Implementação de um Método para Cálculo das Frações de Área de Componentes Homogêneos de uma Imagem usando Análise Linear de Histogramas**. Santa Maria: Curso de Ciência da Computação. Universidade Federal de Santa Maria., 2007.

GAMMA, E.; HELM, R.; R., J.; VLISSIDES, J. **Padrões de Projeto. Soluções Reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2005.

HAMU, D. Enterprise Frameworks. In: BUILDING APPLICATION FRAMEWORKS: OBJECT-ORIENTED FOUNDATIONS OF FRAMEWORK DESIGN, 1999, New York. **Anais...** John Wiley & Sons, 1999. p.83–87.

LIANG, S. **The Java™ Native Interface - Programmer's Guide and Specification**. [S.l.]: Addison-Wesley, 1999.

MAZZANTI, E. S. **Técnicas de Reengenharia de Software aplicadas em uma ferramenta de representação 3D da distribuição de cores de imagens digitais.** Santa Maria: Curso de Ciência da Computação. Universidade Federal de Santa Maria., 2008.

NVIDIA. **NVIDIA CUDA Programming Guide**. Version 2.1.ed. [S.l.: s.n.], 2008.

PARNAS, D.; CLEMENTS, P.; WEISS, D. The Modular Structure of Complex Systems. **IEEE Transactions on Software Engineering**, [S.l.], n.SE-11, p.259–266, 1985.

PREE, W. **Design Patterns for Object-Oriented Software Development**. [S.l.]: Addison-Wesley, 1995. (ACM Press).

PREE, W. Hot-Spot-Driven Development. In: **Building Application Frameworks**: object-oriented foundations of framework design. New York: John Wiley & Sons, 1999.

SCHMIDT, D. C.; GOKHALE, A.; NATARAJAN, B. Leveraging application framework. **ACM Press**, [S.l.], v.Vol. 2 Issue 5, p.66–75, 2004.

SUN, M. **Core J2EE Patterns - Data Access Object**. Disponível em http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html, Acesso em: Junho de 2009.

SUN, M. **Independent Tests Demonstrate Write Once Run Anywhere Abilities of Java**. Disponível em http://www.sun.com/smi/Press/sunflash/1997-09/sunflash.970918.2.xml, Acesso em: Março 2008.

SUN, M. **Trail**: the reflection api. Disponível em http://java.sun.com/docs/books/tutorial/reflect/index.html, Acesso em: Abril de 2009.

SUN, M. **JavaBeans Concepts**. Disponível em http://java.sun.com/docs/books/tutorial/javabeans/whatis/index.htm, Acesso em: Abril de 2009.

SUN, M. **Java ME Platform Overview**. Disponível em http://java.sun.com/javame/technology/index.jsp, Acesso em: Junho de 2009.

SUN, M. **Java Advanced Imaging (JAI) API**. Disponível em http://java.sun.com/javase/technologies/desktop/media/jai/, Acesso em: Maio de 2009.

VITHARANA, P. Risks and challenges of component-based software development. **Communication of the ACM**, [S.l.], 2003.

W3C. **Extensible Markup Language (XML)**. Disponível em http://www.w3.org/XML/, Acesso em: Maio de 2009.

WELFER, D. **Padrões de projeto no desenvolvimento de sistemas de processamento de imagens**. 2004. Dissertação (Mestrado) — Dissertação de Mestrado do PPGEP-UFSM, Santa Maria.

WOLFINGER, R. Plug-in Architecture and Desing Guidelines for Customizable Enterprise Applications. **OOPSLA'08**, [S.l.], 2008.