ARTIGOS CONSEGI 2009

CONGRESSO INTERNACIONAL
SOFTWARE LIVRE E
GOVERNO ELETRÔNICO

MINISTÉRIO DAS RELAÇÕES EXTERIORES

Ministro de Estado Embaixador Celso Amorim
Secretário-Geral Embaixador Samuel Pinheiro Guimarães

FUNDAÇÃO ALEXANDRE DE GUSMÃO

Presidente Embaixador Jeronimo Moscardo

MINISTÉRIO DA FAZENDA

Ministro de Estado Guido Mantega

SERVIÇO FEDERAL DE PROCESSAMENTO DE DADOS

Presidente Marcos Vinícius Ferreira Mazoni

A *Fundação Alexandre de Gusmão*, instituída em 1971, é uma fundação pública vinculada ao Ministério das Relações Exteriores e tem a finalidade de levar à sociedade civil informações sobre a realidade internacional e sobre aspectos da pauta diplomática brasileira. Sua missão é promover a sensibilização da opinião pública nacional para os temas de relações internacionais e para a política externa brasileira.

Ministério das Relações Exteriores
Esplanada dos Ministérios, Bloco H
Anexo II, Térreo, Sala 1
70170-900 Brasília, DF
Telefones: (61) 3411-6033/6034
Fax: (61) 3411-9125
Site: www.funag.gov.br

Artigos CONSEGI 2009

Congresso Internacional
Software Livre e Governo Eletrônico

Brasília, 2009

Equipe Técnica:
Eliane Miranda Paiva
Maria Marta Cezar Lopes
Cíntia Rejane Sousa Araújo Gonçalves
Erika Silva Nascimento

Programação Visual e Diagramação:
Juliana Orem e Maria Loureiro

Impresso no Brasil 2009

Congresso Internacional Software Livre e Governo Eletrônico (I : 2009 : Brasília)
Artigos CONSEGI 1009. - Brasília : Fundação Alexandre Gusmão, 2009.
176p.

ISBN: 978-85-7631-164-5

1. Computadores - Programas. 2. Administração Pública - Informática. 3 Governo - Informática. I. Título.

CDU 004.4
CDU 004:35

Fundação Alexandre de Gusmão
Ministério das Relações Exteriores
Esplanada dos Ministérios, Bloco H
Anexo II, Térreo
70170-900 Brasília – DF
Telefones: (61) 3411 6033/6034
Fax: (61) 3411 9125
Site: www.funag.gov.br
E-mail: funag@mre.gov.br

Depósito Legal na Fundação Biblioteca Nacional conforme Lei n° 10.994, de 14/12/2004.

Sumário

Prefácio

"(...) Agora que o prato está feito, é muito fácil a gente comer. Mas fazer esse prato não foi brincadeira. Eu lembro da primeira reunião que nós fizemos, na Granja do Torto, em que eu entendia absolutamente nada da linguagem que esse pessoal decidia, e houve uma tensão imensa entre aqueles que defendiam a adoção do Brasil do software livre e aqueles que achavam que nós deveríamos fazer a mesmice de sempre, ficar do mesmo jeito, comprando, pagando a inteligência dos outros e, graças a Deus, prevaleceu no nosso país a questão e a decisão do software livre. Nós tínhamos que escolher: ou nós íamos para a cozinha preparar o prato que nós queríamos comer, com os temperos que nós queríamos colocar e dar um gosto brasileiro na comida, ou nós iríamos comer aquilo que a Microsoft queria vender para a gente. Prevaleceu, simplesmente, a ideia da liberdade.(...)

O software livre é um pouco isso, ou seja, é dar às pessoas a oportunidade de fazer coisas novas, de criar coisas novas, de valorizar a individualidade das pessoas. Porque não tem nada que garanta mais a liberdade do que você garantir a liberdade individual, que as pessoas permitam aflorar a sua criatividade, a sua inteligência, sobretudo em um país novo como o Brasil, em que a criatividade do povo possivelmente seja, sem nenhum menosprezo a outros povos, o povo de maior criatividade no século XXI. (...)

Porque esse país ainda está se encontrando consigo mesmo, porque durante séculos nós éramos tratados como se fossemos cidadãos de terceira classe, nós tínhamos que pedir licença para fazer as coisas, nós só podíamos fazer as coisas que os Estados Unidos permitissem, ou se a Europa permitisse. E a nossa auto estima está em alta. Nós aprendemos a gostar de nós mesmos. Nós estamos descobrindo que nós podemos fazer as coisas. Nós estamos descobrindo que ninguém é melhor do que nós. Pode ser igual, mas melhor não são, não têm mais criatividade do que nós. O que nós precisamos é oportunidade. (...)

Eu só queria dizer para vocês uma coisa: olhem, eu tenho mais um ano e meio de mandato. Mais um ano e meio de mandato. É importante que vocês detectem aquilo que nós já fizemos e que precisa ser aperfeiçoado. E é preciso que vocês detectem aquilo que nós ainda não conseguimos fazer, e nos ajudem a fazer. (...)

Nós não sabemos tudo, nós sabemos apenas uma parte. Sozinho talvez você também não saiba tudo, saiba só uma parte. Mas se a gente juntar um pouco do que cada um de vocês sabe, a gente possa construir um tudo que falta para a gente, definitivamente, democratizar este país de verdade, e que todos sejam livres e que possam fazer as coisas bem. As pessoas de bem são maioria. Não vamos ficar nervosos porque de vez em quando aparece um maluco falando as coisas. Tem até um site propondo morte ao Lula. Não tem problema, os que propõem vida são infinitamente maiores. Infinitamente maiores.

Então, eu queria dizer para vocês que entrar naquele "corredor polonês" e ver aquela gama extraordinária de meninos e meninas, acho que todos com menos de 25, 30 anos de idade, é a gente poder sair daqui e dizer em alto e bom som: "Finalmente este país se encontrou consigo mesmo. Finalmente este país está tendo o gosto da liberdade de informação". (...)

Presidente Luiz Inácio Lula da Silva
Parte do discurso proferido no
Fórum Internacional de Software Livre – FISL
Porto Alegre, 26 de junho de 2009

Apresentação

Liberdade é mais do que voar.
É criar e compartilhar!

Em 1906, um brasileiro surpreendeu o mundo. Alberto Santos Dumont voou. A bordo de um aparelho mais pesado que o ar, denominado 14-bis, percorreu um curto trajeto a uma altura de dois metros diante de milhares de espectadores franceses. Esse pequeno-grande vôo tornou-o o "pai da aviação". No entanto, sua contribuição à humanidade foi ainda maior. Ao ser pressionado pelo governo francês para que patenteasse seu invento, Santos Dumont resolveu publicar as especificações de sua descoberta em todas as revistas científicas da época. Ele trocou os royalties pelo progresso da Ciência. E, sem saber, sua atitude altruísta tornou-se um dos primeiros exemplos de licença GPL (Genral Public License) do mundo. No artigo "Licenças de Software Livre: História e Classificação", que faz parte desta coletânea, os pesquisadores da USP, Vanessa Sabino e Fábio Kon, tratam da importância das licenças livres.

Santos Dumont era um homem muito a frente de seu tempo. Ao agregar diversos conhecimentos da época e ao tornar pública sua invenção sem nenhum tipo de restrição, ele praticou há 100 anos atrás algo que está na vanguarda dos dias atuais: o conhecimento livre. Numa sociedade em que a comunicação acontece cada vez mais em rede (Internet) e a produção de conteúdos está aberta a todos os indivíduos (Web 2.0), compartilhar o conhecimento e trocar experiências estão

na ordem do dia. Um dos exemplos bem-sucedidos desta situação é o Consórcio OW2, comunidade global, integrada por instituições européias, asiáticas e americanas, que tem como objetivo o desenvolvimento de middleware - componentes flexíveis e adaptáveis de software - para facilitar a interação entre produtores de softwares abertos e os consumidores destes produtos. O artigo "Platform Strategies and the OW2 Consortium" de Cedric Thomas, CEO do consórcio, fala sobre a melhor estratégia para conquistar espaço neste cenário.

Com um novo panorama, a produção de tecnologia sofre profundas mudanças e o software livre (SL) faz parte delas. Surgido na década de 80, este movimento com o objetivo de criar programas de computador que podem ser usados, copiados, estudados, modificados e redistribuídos sem nenhuma restrição ganhou força. A liberdade de suas diretrizes, por serem intrínsecas ao conceito, e sua característica comunitária, ou seja, baseada em comunidades de discussão e desenvolvimento, garantem inúmeros benefícios com relação ao software proprietário como a economia de recursos, a celeridade nas soluções de problemas e, principalmente, a independência tecnológica. "O software livre representa a oportunidade de fazer coisas novas, de criar coisas novas, de valorizar a individualidade das pessoas", afirmou o presidente Luís Inácio Lula da Silva, em seu discurso durante o 10° Fórum Internacional de Software Livre cuja transcrição esta no prefácio desta coletânea.

As principais vantagens constatadas com a adoção do software livre são a possibilidade de investir-se em inteligência tecnológica nacional e a disseminação nas instituições governamentais da lógica de cooperação e compartilhamento, tornando o governo eletrônico brasileiro mais eficiente, ágil e justo. No artigo "RekZit – vencendo os desafios da captura e gerenciamento de requisitos com uma ferramenta livre ", os analistas de TI Leonardo Thomas Torres Santos, Márcio Lima Albuquerque e Sandro de Carvalho Franco falam sobre um software livre que pode ajudar na ampliação desta relação.

O Serviço Federal de Processamento de Dados – Serpro, empresa pública do Ministério da Fazenda responsável pelo desenvolvimento de soluções como o Sistema de Declaração do Imposto de Renda - Receitanet, caminha há muito tempo neste sentido. Por ter amadurecido

ao longo dos últimos seis anos o uso e o desenvolvimento de SL, o Serpro criou uma extensa gama de prestação de serviços baseadas em plataforma aberta. O Expresso, por exemplo, é uma suíte de comunicação com 20 mil usuários que integra diversas facilidades como correio eletrônico e agenda. Ele foi desenvolvido em parceria com a Companhia de Informática do Paraná – Celepar e, atualmente, mantem uma comunidade que envolve outras instituições públicas como a Itaipu Binacional. Outro exemplo é o framework de desenvolvimento Demoiselle. Tal ferramenta livre, cujo nome homenageia o mais bem-sucedido projeto de Santos Dumont, trabalha com a lógica de blocos de construção de programas, codificando e reutilizando conhecimento, integrando diferentes tecnologias e viabilizando maior controle na elaboração de soluções. O objetivo é convergir os produtos de TI utilizados pelo Governo Federal para que haja otimização de recursos, celeridade na produção e a apropriação do conhecimento. Esta plataforma é tema do artigo "Padronização do desenvolvimento de aplicações corporativas por meio de um arcabouço de software baseado em uma arquitetura de referência", de Flávio Lisboa, analista do Serpro.

O software livre é a alternativa viável para que os diferentes níveis de governo possam atender às expectativas de seus cidadãos sem estarem aprisionados a tecnologias proprietárias e monopolistas. Muito mais que economia aos cofres públicos, o uso do conhecimento livre pode trazer desenvolvimento social e transformar o que seria gasto com pagamento de royalties e licenças em investimento nas pessoas. O II Congresso Software Livre e Governo Eletrônico – Consegi 2009 irá tratar desse e de outros assuntos. Santos Dumont há um século acreditava na lógica do compartilhamento e fez o homem voar. O objetivo é que o Consegi seja terreno fértil para vôos cada vez mais livres e audaciosos. E o limite é a própria capacidade humana de criar.

Marcos Vinicius Ferreira Mazoni
Diretor-Presidente do Serpro e
Coordenador do Comitê de Implementação do
Software Livre no Governo Federal

Licenças de Software Livre: História e Classificação[1]

Vanessa Sabino e Fabio Kon
{bani,kon}@ime.usp.br
Centro de Competência em Software Livre
Departamento de Ciência da Computação - IME - USP
ccsl.ime.usp.br

1. Introdução

Embora a questão das licenças seja muito importante para o desenvolvimento e adoção do software livre, muitas vezes desenvolvedores e instituições usuárias não dão a devida importância a esse tema. Programas de software livre em geral são de fácil acesso. Porém, a simples obtenção de um programa não significa que a pessoa pode fazer o que quiser com ele. As licenças de software livre são documentos através dos quais os detentores dos direitos sobre um programa de computador autorizam usos de seu trabalho que, de outra forma, estariam ditados unicamente pelas leis vigentes no local.

Além do uso como usuário final, esses usos autorizados permitem que desenvolvedores possam adaptar o software para necessidades mais específicas, utilizá-lo como fundação para construção de programas mais complexos, entre diversas outras possibilidades. Neste capítulo, após um breve histórico e contextualização do software livre, apresentamos uma classificação dos principais tipos de licenças e veremos como a escolha da licença influencia a forma como o software poderá ser usado, desenvolvido e distribuído.

[1] Pesquisa apoiada pelo CNPq e pelo Projeto Qualipso (*European Comission*).

2. Breve Histórico do Software Livre

Com o surgimento dos primeiros computadores vendidos comercialmente, a partir da década de 1950, foram criados também os primeiros programas que iriam ser executados neles. Muitas vezes ocorria uma venda casada entre hardware e software, pois os programas eram fortemente acoplados à arquitetura das máquinas em que eram executados. Nessa época, o foco das empresas era na venda do hardware e não eram colocadas muitas restrições no uso que as pessoas fariam do software [CK08]. Elas podiam adaptá-lo como quisessem, de forma a fazer melhor uso do harware que tinham disponível, sem sofrer repreensões.

Na década de 1970 a situação começou a se modificar. Algumas empresas, como a Microsoft, não estavam satisfeitas com a forma como seus programas eram redistribuídos sem que a empresa recebesse *royalties* pelas cópias. Assim, em 3 de fevereiro de 1976, Bill Gates escreveu a *Open Letter to Hobbyists*, que foi publicada na *newsletter* do *Homebrew Computer Club*. Nessa carta, Bill Gates afirma que o total de *royalties* recebidos pelo Altair BASIC era equivalente a apenas dois dólares por hora gasta em seu desenvolvimento e documentação. Ele ainda alega que a prática de compartilhamento de software não é justa e afirma que tal prática evita que software bem feito seja escrito. Assim, nessa época começou uma mudança de postura na indústria, que passou a proibir que o software fosse copiado ou modificado. Surgiu então o que chamaremos de *software fechado*, caracterizado pelas restrições que são feitas à forma como ele será utilizado.

Como resposta a essa nova situação, surgiram iniciativas voltadas para retomar a liberdade de melhorar e compartilhar o software.

2.1. O Projeto GNU e o Software Livre

Em 27 de setembro de 1983, Richard Stallman postou uma mensagem nos *newsgroups* net.unix-wizards e net.usoft com o assunto "*new Unix implementation*". Nessa mensagem, ele informa que está começando a escrever um sistema compatível com UNIX chamado GNU (um acrônimo recursivo para *Gnu's Not Unix*) e que ele será dado a todas as pessoas interessadas. Ele cita alguns componentes que seriam incluídos, tais como núcleo do sistema operacional, compilador C e editor de texto, e propõe algumas melhorias em relação aos sistemas UNIX existentes na época. Ele também explica na mensagem o motivo

dele precisar escrever o GNU: segundo seus princípios, se ele gosta de um programa ele precisa compartilhá-lo com outras pessoas que também gostem dele. Para continuar usando computadores sem violar seus princípios, ele decidiu criar um conjunto suficiente de software livre para que ele pudesse prosseguir sem usar qualquer software que não fosse livre. Para finalizar a mensagem, ele pede contribuições na forma de máquinas, ajuda para escrever o software e dinheiro. No livro *Open Sources* [DOS99], Stallman conta um pouco mais sobre a história do projeto, cujos principais fatos são discutidos abaixo.

No início de 1984 Stallman largou seu emprego no laboratório de Inteligência Artificial do MIT, para garantir que ele teria os direitos sobre o trabalho, e começou a dedicar-se em tempo integral ao projeto. O primeiro programa criado foi o GNU Emacs, que foi disponibilizado por FTP. Porém, como naquela época muitas pessoas não tinham acesso à Internet, Stallman também começou a ganhar dinheiro vendendo cópias físicas do programa, em fita magnética, por 150 dólares. Ele considera esse negócio de distribuição de software livre um precursor das empresas que hoje distribuem o GNU/Linux. Pouco tempo depois foi desenvolvido o GCC (que na época significava *GNU C Compiler*, mas atualmente, com a adição de outras linguagens, é chamado de *GNU Compiler Collection*), que até hoje é um dos componentes mais importantes do sistema GNU. Ao longo dos anos, Stallman decidiu incorporar, ao sistema GNU, software que não foi escrito pelo projeto GNU, como por exemplo o Linux e o X Window System. Isso era possível pois esses programas também eram livres.

Enquanto o sistema GNU era desenvolvido, também foi sendo formado o conceito de *Free Software*, ou Software Livre, levando à criação da *Free Software Foundation* por Stallman em 1985. Segundo sua definição, um software é livre se:

(1) você tem a liberdade de executar o programa, para qualquer propósito;

(2) você tem a liberdade de modificar o programa para adaptá-lo às suas necessidades (para tornar essa liberdade efetiva na prática, você precisa ter acesso ao código-fonte, já que fazer alterações em um programa sem ter o código-fonte é muito difícil);

(3) você tem a liberdade de redistribuir cópias gratuitamente ou mediante pagamento e

(4) você tem a liberdade de distribuir versões modificadas do programa para que a comunidade possa se beneficiar de suas melhorias.

Como o objetivo do projeto GNU era garantir essas liberdades para os usuários, foi criado um sistema de distribuição chamado *copyleft*, que visava impedir que o software se tornasse fechado. Mais detalhes a respeito disso serão vistos na Seção 4.2.

2.2. O Movimento *Open Source*

Segundo Eric Raymond, devido ao caráter ético e político do movimento proposto pela *Free Software Foundation*, muitas empresas viam o software livre como "anti-capitalista" e eram relutantes em adotá-lo. Observando isso, ele teve a ideia de mudar a abordagem como seria apresentado o software livre para pessoas mais conservadoras e criou o termo *Open Source* em 1997. Não usando a palavra *free*, Raymond estava não apenas evitando a confusão com gratuito, como também tirando a conotação esquerdista do termo proposto por Stallman.

Também em 1997, Bruce Perens havia escrito o *Debian Free Software Guidelines* para definir o que seria aceito como software livre pela distribuição Debian, dado que havia outras licenças além daquelas propostas pela *Free Software Foundation* que alegavam serem livres. Então, Eric Raymond contatou Bruce Perens para discutir a ideia de *Open Source* e decidiram a partir daí adaptar o documento *Debian Free Software Guidelines* para formar a *Open Source Definition*, ou Definição de Código Aberto. Eles registraram a marca *Open Source* e formaram a *Open Source Initiative* (OSI) [DOS99].

No início de 1998 tivemos o primeiro caso de uma empresa já consolidada no mercado abrir o código de seu software: o Netscape. Junto com Eric Raymond, os executivos da Netscape escreveram uma licença que adotava alguns princípios de *copyleft*, mas que permitia à Netscape continuar distribuindo versões fechadas com o código aberto do navegador. Em seguida, Raymond publicou um pedido à comunidade intitulado *Goodbye, "Free Software"; Hello, "Open Source"*, em que insistia que o termo *open source* era melhor do que *free software* e devia ser adotado.

Com essa nova abordagem, que ressaltava os benefícios técnicos decorrentes da metodologia adotada pela comunidade, e tendo como exemplo a Netscape, a adoção do software livre por parte das empresas sofreu um grande impulso. Porém, a maior disseminação do software livre

nesses termos não deixou Stallman satisfeito. Segundo ele, se as pessoas não introjetam a liberdade associada ao software livre, elas voltarão a usar software fechado quando este apresentar vantagens práticas.

Stallman também argumenta que a expressão "código aberto" tem como significado óbvio simplesmente "você pode olhar para o código", o que é um critério muito mais fraco do que a definição oficial de código aberto, que pode ser vista a seguir em tradução livre do documento disponível em opensource.org/docs/osd.

Definição de Código Aberto: Introdução - Código aberto não significa apenas acesso ao código-fonte. Os termos de distribuição do software de código aberto devem estar de acordo com os seguintes critérios:

1. Redistribuição Livre - A licença não deve restringir qualquer das partes de vender ou doar o software como um componente de uma distribuição agregada de software, contendo programas oriundos de várias fontes diferentes. A licença não deve exigir *royalties* ou qualquer outro tipo de pagamento para venda.

2. código-fonte - O programa deve incluir o código-fonte, e deve permitir a distribuição na forma de código-fonte bem como na forma compilada. Quando alguma forma do produto não é distribuída com o código-fonte, é necessário haver meios bem divulgados para obtenção do código por não mais que um custo razoável de reprodução, preferencialmente através de download pela Internet gratuitamente. O código-fonte deve ser a forma preferencial pela qual um programador alteraria o programa. código-fonte obscurecido deliberadamente não é permitido. Formas intermediárias, como a saída de um processador ou tradutor, não são permitidas.

3. Trabalhos Derivados - A licença deve permitir modificações e trabalhos derivados e precisa permitir que eles sejam distribuídos sob os mesmos termos da licença do software original.

4. Integridade do código-fonte do Autor - A licença pode restringir a distribuição de código-fonte em forma modificada somente se a licença permitir a distribuição de "arquivos de *patch*" com o código-fonte para o propósito

de modificar o programa em tempo de compilação. A licença deve permitir explicitamente a distribuição do software compilado a partir de um código modificado. A licença pode exigir que trabalhos derivados usem um nome ou número de versão diferentes do original.

5. Sem Discriminação a Pessoas ou Grupos - A licença não deve discriminar qualquer pessoa ou grupo de pessoas.

6. Sem Discriminação a Áreas de Empreendimento - A licença não deve restringir qualquer pessoa a fazer uso do programa em uma área de empreendimento específica. Por exemplo, ela não pode restringir o uso do programa comercialmente ou o uso em pesquisas genéticas.

7. Distruibuição da Licença - Os direitos associados ao programa devem ser aplicáveis a todos para quem o programa é redistribuído, sem a necessidade de execução de licenças adicionais para essas partes.

8. A Licença não deve ser específica a um produto - Os direitos associados ao programa não devem depender dele ser parte de uma distribuição específica de software. Caso o programa seja extraído dessa distribuição e usado ou distribuído nos termos da licença do programa, todas as partes para as quais o programa é redistribuído devem ter os mesmos direitos que aqueles concedidos em conjunto com a distribuição de software original.

9. A Licença não deve restringir outro software - A licença não deve colocar restrições em outro software que seja distribuído junto com o software licenciado. Por exemplo, a licença não deve exigir que todos outros programas distribuídos no mesmo meio sejam software de código aberto.

10. A Licença deve ser neutra às tecnologias - Nenhuma condição da licença deve ser estabelecida em uma tecnologia individual específica ou estilo de interface.

Baseando-se na Definição de Código Aberto, a OSI aprova as licenças que podem ser consideradas *open source*. Para isso, as licenças passam por um processo público de revisão para assegurar que estão em conformidade

com as normas e expectativas da comunidade. A OSI conta com uma lista de mais de 70 licenças aprovadas (www.opensource.org/licenses) e possui um comitê para tratar do assunto de proliferação das licenças, visando criar mecanismos para facilitar a escolha da licença.

3. A Importância do Software Livre

Estima-se que hoje há centenas de milhões de usuários de software livre no mundo; se considerarmos também usuários indiretos, que usam serviços baseados em software livre, como Google, Amazon ou eBay, esse número deve ser superior a 1 bilhão. Neste capítulo, veremos as principais vantagens e desvantagens deste modelo, além das perspectivas de longo prazo.

3.1. Vantagens do Software Livre

A *Free Software Foundation* e a *Open Source Initiative* apresentam justificativas diferentes para o uso do software livre. Neste relatório, focaremos no discurso da *Open Source Initiative*, não discutindo em profundidade as questões éticas relacionadas ao software livre que são defendidas pela *Free Software Foundation*.

A principal vantagem do software livre é permitir o compartilhamento do código-fonte. Como consequência desse compartilhamento, evita-se a duplicação de esforços quando mais de uma entidade está interessada no desenvolvimento de uma aplicação com funcionalidade similares, reduzindo assim o custo do desenvolvimento.

Além disso, autores como Eric Raymond [Ray01] afirmam que software livre tem condições de ter maior qualidade do que seus equivalentes fechados. Uma das justificativas para essa afirmação de Raymond é conhecida como “A Lei de Linus”, que diz “dados olhos suficientes, todos os *bugs* são superficiais”. Isso significa que, com o maior número de usuários que tem acesso ao programa e até ao código-fonte, o software é testado melhor e os problemas existentes no código são encontrados mais rapidamente. Outro fator que contribui para a qualidade é o orgulho pessoal do desenvolvedor, pois a partir do momento que seu código poderá ser lido por mais pessoas é esperado que ele seja mais cuidadoso com seu trabalho. A competição também é facilitada no software livre e, assim, se o grupo original de desenvolvedores não está fazendo um bom trabalho para

manter o projeto, é possível que um novo grupo faça um *fork* para suprir as deficiências.

No Brasil, caracterizado por um mercado local, onde apenas uma parte inexpressiva da indústria de software usa o modelo de desenvolvimento de software fechado para venda do produto sem customizações, o software livre pode trazer ainda mais benefícios. Ao desenvolver serviços e soluções baseadas em software livre, as empresas brasileiras podem deixar de investir em licenças pagas promovendo a evasão de divisas e contribuindo negativamente para a balança comercial brasileira.

Para os usuários também é vantagem o software livre, pois evita a dependência de um fornecedor. Isso traz tanto uma vantagem financeira, dado que normalmente é necessário pagar por novas versões do sistema quando o software é fechado, como também maior liberdade para o usuário, que pode adaptar o software para suas necessidades. É possível corrigir falhas de segurança e *bugs*, escrever uma documentação melhor ou contratar uma empresa que faça isso independentemente de quem seja o autor original [Web04]. Além disso, se o fornecedor original abandona o projeto, no caso do software fechado não há nada que possa ser feito para continuar o desenvolvimento do projeto, enquanto que no software livre é possível que outro grupo adote o projeto e continue a evoluir o código.

3.2. Desvantagens do Software Livre

Também podemos levantar algumas desvantagens do software livre em relação a alternativas fechadas. Um dos principais motivos que leva uma empresa a optar por um software fechado quando há um similar livre disponível é a ausência de garantias e suporte desse último. As licenças de software livre em geral eximem o autor de qualquer responsabilidade tanto quanto é permitido pelas leis do local. Dessa forma, em casos em que a empresa precisa fornecer garantias aos seus clientes, ou quando a indisponibilidade de um sistema pode causar grandes prejuízos, pode ser melhor que a empresa adquira uma solução em que eventuais problemas sejam delegados a um fornecedor ou que esse tenha que indenizar a empresa. Porém, é importante deixar claro que, apesar das licenças de software livre normalmente incluírem cláusulas sobre a ausência de responsabilidades, há casos em que empresas optam por fornecer, como um serviço, garantias e/ou suporte para um determinado software livre. Além disso, grande parte do software fechado disponível

também busca em seu contrato se eximir de responsabilidades tanto quanto a legislação permite.

Qualidade, reputação e imagem também são vistos como desvantagens na adoção do software livre. Quando não há uma empresa de renome por trás do software oferecido, há uma maior dificuldade em avaliar as alternativas, além de um receio de que o produto seja abandonado e deixe-se de oferecer suporte para ele. Também influi negativamente na sua imagem o fato do software estar disponível gratuitamente.

Já do ponto de vista de quem produz software, optar pelo modelo aberto pode ser visto como desvantagem à medida que a propriedade intelectual está exposta. Como os concorrentes têm fácil acesso ao código-fonte, é necessário que a empresa criadora do software mantenha seu diferencial para garantir seu mercado a longo prazo. Caso contrário, o software pode tornar-se um *commodity* e as possibilidades de lucro da empresa são reduzidas [Web04]. Por outro lado, no caso de uma licença que obriga que as melhorias sejam disponibilizadas como software livre, é mais difícil que uma nova empresa adquira vantagem competitiva sobre outra que já está consolidada no mercado, pois qualquer diferencial no produto pode ser prontamente absorvido pela empresa líder [DOS99].

Finalmente, observa-se também que o modelo de negócio tradicional de vender software da mesma forma como se vende qualquer outro bem material não funciona bem. É necessário buscar outros modelos de negócio.

3.3. Perspectivas do Software Livre

O modelo do software livre possui características bastante interessantes para o mercado de Tecnologia da Informação. Há uma certa ambiguidade na relação entre clientes, empresas e a comunidade de desenvolvimento de software, permitindo novos modelos de negócio. Os consumidores têm a possibilidade de também contribuir para o bem coletivo, agregando valor. Porém, a parcela de consumidores que realmente contribui é muito pequena, e os demais requerem uma quantidade desproporcional de serviços de suporte, que não são facilmente escaláveis. A competição com produtos fechados, que já arrecadam dinheiro com a venda do software, empurra para baixo o valor do suporte, dificultando a situação das empresas que têm como modelo a venda de serviços baseados em software livre [Web04].

Em um recente estudo da Forrester [Con08] sobre o uso de software livre e seu impacto na indústria de software, foram observadas as seguintes tendências:

- crescimento da adoção de software livre pelo usuário final, na forma de ferramentas de produtividade e aplicações de negócio;
- diferenças na adoção de software livre de acordo com ramo de atividade, com fábricas adotando fortemente o software livre em sua infraestrutura enquanto serviços financeiros usam o software livre em aplicações de mais alto nível;
- uso de software livre em aplicações de missão crítica, serviços e produtos;
- altos índices de satisfação em relação a custo e qualidade do software livre;
- diversidade de critérios como motivadores para adoção de software livre, destacando custo, independência, flexibilidade e inovação;
- adoção, por parte das empresas, das boas práticas e princípios da comunidade de software livre;
- crescimento de provedores de serviço e centros de competência para prover suporte científico-tecnológico e também comercial para produtos de software livre.

Hoje em dia, o software livre aparece na maior parte das projeções sobre o futuro do software. Porém, é importante ressaltar que cada novo paradigma que surge na indústria altera substancialmente as perspectivas, tornando muito difícil prever como estará o mercado de software daqui a alguns anos [CK08].

4. Famílias de Licenças

O detentor dos direitos patrimoniais sobre um software, quando decide torná-lo livre, deve escolher os termos em que seu trabalho será distribuído, ou seja, os direitos que ele estará transferindo para as outras pessoas e quais as condições que serão aplicadas. O documento que formaliza esse ato é a licença, que normalmente é distribuída junto com o código-fonte.

Há inúmeras possibilidades para redigir o texto de uma licença de software livre, mas a prática mais comum é reaproveitar alguma das licenças já

consolidadas na comunidade. Dessa forma, reduz-se a proliferação de licenças, que deve ser evitada pois gera trabalho adicional para os usuários, uma vez que torna-se necessário para eles estudar os termos de cada nova licença presente no software que irão utilizar. No portal SourceForge, que hospeda um grande número de projetos de código aberto, ao criar um novo projeto são apresentadas oito opções de licenças, que são as mais utilizadas naquele repositório, além da opção de domínio público: GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL), BSD License, MIT License, Apache License V2.0, Artistic License, Mozilla Public License 1.1 (MPL 1.1) e Academic Free License (AFL). Além disso, no final da página que lista as licenças, há um ponteiro para visualização de todas as licenças disponíveis, contendo, em outubro de 2008, 74 opções. Há ainda a possibilidade do usuário incluir sua própria licença.

A *Open Source Initiative* (OSI) mantém uma lista de licenças aprovadas de acordo com a definição de código aberto (vide Seção 2.2), organizadas em diversas categorias, além de uma lista de licenças atualmente buscando aprovação. A OSI também está trabalhando com um comitê (anti-)proliferação de licenças, visando separar e dar destaque a algumas licenças que passarão a ser *recomendadas*, em oposição às meramente *aprovadas*.

Considerando que uma das questões mais importantes é a compatibilidade entre licenças, vamos separar as licenças descritas aqui em três categorias, de acordo com a presença de termos que impõem restrições de licenciamento na redistribuição do trabalho ou criação de trabalhos derivados. Em relação a esta característica, as licenças são consideradas *permissivas* ou *recíprocas*, sendo que entre as recíprocas devemos ainda considerar que algumas forçam que seja mantida a mesma licença em mais casos do que outras, e assim as dividimos entre recíprocas parciais, que também recebem a denominação de *copyleft* fraco, e recíprocas totais.

4.1. Permissivas

As licenças permissivas, também chamadas de licenças acadêmicas por alguns autores, como Rosen [Ros05] e Laurent [Lau04], em referência às origens das licenças BSD (*University of California, Berkeley*) e MIT (*Massachusetts Institute of Technology*), impõem poucas restrições às pessoas que obtém o produto. Muitas vezes, tais licenças são usadas em projetos de pesquisa de universidades, que servem como prova de conceito

de alguma tecnologia que poderá ser explorada comercialmente no futuro. No caso das licenças permissivas, não é feita nenhuma restrição ao licenciamento de trabalhos derivados, estes podendo inclusive serem distribuídos sob uma licença fechada. Nesta seção serão discutidas as licenças BSD, MIT e Apache.

Licenças permissivas são uma ótima opção para projetos cujo objetivo é atingir o maior número de pessoas, não importando se na forma de software livre ou de software fechado. Temos vários exemplos desse modelo no BSD Unix, que continha o software de TCP/IP que hoje é usado na maior parte das implementações desse protocolo, incluindo a da Microsoft [Lau04]. Outro exemplo é o BIND, Berkeley Internet Name Daemon, cuja implementação livre é até hoje usada nos principais servidores de DNS, apesar de haver também várias implementações fechadas. Segundo Laurent [Lau04], há bilhões de dólares em atividade econômica associada apenas com a pilha de software para Internet originalmente liberada sob a licença BSD.

Alguns argumentam que o uso desse tipo de licença não incentiva o modelo de software livre, pois empresas se aproveitam da comunidade para desenvolver software que será fechado. Porém, quando são usadas licenças permissivas, em geral os autores estão cientes dessa possibilidade e não vêem isso como um problema. Um caso conhecido em que, de fato, os autores se arrependeram da licença que adotaram é o do Kerberos, desenvolvido no MIT, que posteriormente foi adotado pela Microsoft, que desenvolveu extensões fechadas [Lau04]. Mas o mais provável, caso a licença não permitisse isso, seria que a Microsoft adotasse algum outro sistema de segurança e o Kerberos não se tornaria tão popular.

Por outro lado, em alguns casos, não é necessário que haja restrições na licença para garantir a continuidade do modelo de desenvolvimento aberto, como no exemplo do servidor Apache. Duas características explicam o domínio do servidor Web livre no mercado, segundo Laurent [Lau04]: a marca forte, cujo uso é protegido pela própria licença do Apache, e a importância da conformidade com os padrões, que evita a disseminação de extensões fechadas.

4.1.1. A Licença BSD

A primeira licença que iremos estudar é a BSD (**www.creativecommons.org/licenses/BSD/ legalcode**), que foi a primeira licença de software livre escrita e até hoje é uma das mais usadas. Criada

originalmente pela Universidade da Califórnia em Berkeley para seu sistema operacional derivado do UNIX chamado Berkeley Software Distribution, a licença BSD é usada como modelo por uma ampla gama de software licenciado de modo permissivo.

Os principais motivos que levaram a licença BSD a ser tão difundida são a simplicidade de seu texto e o fato dela ter sido inicialmente adotada por um projeto amplamente disseminado, o que criou um ciclo virtuoso em que mais comunidades a adotaram, tornando-a ainda mais reconhecida.

Porém, originalmente, a licença BSD possuia uma cláusula que determinava que todo material de divulgação relacionado ao software precisava conter uma afirmação que dizia "este produto inclui software desenvolvido pela Universidade da California, Berkeley e seus contribuidores". Tal cláusula, que ficou conhecida como "cláusula de propaganda da licença BSD", não causava muito problema quando era usada apenas no seu contexto original, porém, à medida que outras pessoas e organizações passaram a adotar também a licença BSD, essa cláusula era adaptada para conter outros nomes, e assim surgiam projetos que integravam diversos componentes e determinavam uma longa lista de frases a serem incluídas quando se falava sobre o projeto. Como consequência, em 1999, tal cláusula foi removida, originando a "Licença BSD Simplificada", também conhecida como "Nova Licença BSD" ou "Versão de 3 cláusulas da Licença BSD".

Assim, na versão atualmente utilizada da licença BSD, temos os seguintes termos, apresentados em tradução realizada pela Escola de Direito da Fundação Getúlio Vargas:

> Direitos autorais (C) <ANO>, <TITULAR>
>
> Todos os direitos reservados.
>
> A redistribuição e o uso nas formas binária e código fonte, com ou sem modificações, são permitidos contanto que as condições abaixo sejam cumpridas:
>
> - Redistribuições do código fonte devem conter o aviso de direitos autorais acima, esta lista de condições e o aviso de isenção de garantias subsequente.

- Redistribuições na forma binária devem reproduzir o aviso de direitos autorais acima, esta lista de condições e o aviso de isenção de garantias subsequente na documentação e/ou materiais fornecidos com a distribuição.

- Nem o nome da <ORGANIZAÇÃO> nem o nome dos contribuidores podem ser utilizados para endossar ou promover produtos derivados deste software sem autorização prévia específica por escrito.

Apesar dessa licença utilizar uma linguagem leiga, cujos termos não necessariamente correspondem aos termos encontrados nas leis que visam proteger o autor do software, através deste documento o detentor dos direitos autorais permite que outras pessoas usem, modifiquem e distribuam o software. As únicas exigências são que o nome do autor original não seja utilizado em trabalhos derivados sem permissão, visando proteger sua reputação, dado que o autor pode não ter relação alguma com as modificações realizadas, e no caso de redistribuição do código fonte ou binário, modificado ou não, é necessário que seja mencionado o copyright original e os termos da licença.

É importante ressaltar que a exigência de reproduzir a lista de condições para redistribuição e o aviso legal de isenção de garantias não significa que o trabalho sendo redistribuído precisa estar sob a mesma licença. Além disso, é possível redistribuir o binário sem fornecer o código fonte. Basta que conste na documentação que foi utilizado o software em questão, e que ele foi licenciado nos termos descritos acima.

No final da licença há o aviso legal sobre garantias e responsabilidades apresentado abaixo, que visa proteger o autor de ser processado judicialmente por qualquer insatisfação causada em consequência do uso do software. Porém, tal termo está subordinado às leis locais, que no caso brasileiro, dependendo da relação estabelecida entre o fornecedor e o cliente, não permitem a distribuição de um software com ausência total de garantias.

ESTE SOFTWARE É FORNECIDO PELOS DETENTORES DE DIREITOS AUTORAIS E CONTRIBUIDORES “COMO ESTÁ”, ISENTO DE GARANTIAS EXPRESSAS OU TÁCITAS, INCLUINDO, SEM LIMITAÇÃO, QUAISQUER GARANTIAS IMPLÍCITAS DE COMERCIABILIDADE OU DE ADEQUAÇÃO A

FINALIDADES ESPECÍFICAS. EM NENHUMA HIPÓTESE OS TITULARES DE DIREITOS AUTORAIS E CONTRIBUIDORES SERÃO RESPONSÁVEIS POR QUAISQUER DANOS, DIRETOS, INDIRETOS, INCIDENTAIS, ESPECIAIS, EXEMPLARES OU CONSEQUENTES, (INCLUINDO, SEM LIMITAÇÃO, FORNECIMENTO DE BENS OU SERVIÇOS SUBSTITUTOS, PERDA DE USO OU DADOS, LUCROS CESSANTES, OU INTERRUPÇÃO DE ATIVIDADES), CAUSADOS POR QUAISQUER MOTIVOS E SOB QUALQUER TEORIA DE RESPONSABILIDADE, SEJA RESPONSABILIDADECONTRATUAL, RESTRITA, ILÍCITO CIVIL, OU QUALQUER OUTRA, COMO DECORRÊNCIA DE USO DESTE SOFTWARE, MESMO QUE HOUVESSEM SIDO AVISADOS DA POSSIBILIDADE DE TAIS DANOS.

Por fim, há também versões da licença BSD que removem a terceira condição de redistribuição, relacionada ao autor não endossar trabalhos derivados, restando apenas as duas cláusulas que requerem a reprodução do *copyright* e condições na redistribuição.

Vantagens e Desvantagens.

Uma das principais desvantagens da licença BSD é a quantidade de variantes existentes, pois não fica imediatamente claro para o usuário sob quais termos um software está sendo licenciado quando a única informação que ele recebe é que está sob a licença BSD. Sempre é necessário verificar com mais cuidado, no texto da licença, se o softaware em questão está licenciado pela BSD original, a simplificada, ou a versão sem endosso. Outra desvantagem, do ponto de vista jurídico, é que seus termos são um tanto vagos e não são mapeados diretamente aos termos utilizados no licenciamento de software, havendo um maior esforço de interpretação da licença para demostrar que, de fato, a maioria dos direitos em geral restritos ao autor estão sendo transferidos para o licenciado. Por outro lado, temos como vantagens a simplicidade da licença e sua ampla adoção, que facilitam seu entendimento pelo público geral.

Portanto, o exposto acima faz da licença BSD uma excelente escolha para projetos em que não há uma preocupação a respeito de quais serão os termos que outras pessoas usarão ao redistribuir o trabalho original ou derivado. Um software distribuído sob a licença BSD se aproxima bastante, em relação a seus direitos intrínsecos, de um software em domínio público.

4.1.2. A Licença MIT/X11

A Licença MIT (**www.opensource.org/licenses/mit-license.php**), criada pelo *Massachusetts Institute of Technology*, é também conhecida como Licença X11 ou X, por ter sido redigida para o X Window System, desenvolvido no MIT em 1987.

Essa também é uma licença permissiva e é considerada equivalente à BSD Simplificada sem a cláusula de endosso. Porém, seu texto é bem mais explícito ao tratar dos direitos que estão sendo transferidos, afirmando que qualquer pessoa que obtém uma cópia do software e seus arquivos de documentação associados pode lidar com eles sem restrição, incluindo sem limitação os direitos a usar, copiar, modificar, mesclar, publicar, distribuir, sublicenciar e/ou vender cópias do software. As condições impostas para tanto são apenas manter o aviso de *copyright* e uma cópia da licença em todas as cópias ou porções substanciais do software.

Para finalizar, a licença contém uma cláusula sobre a ausência de garantias e responsabilidades bastante similar à da licença BSD, protegendo os detentores do direito autoral de qualquer processo judicial relacionado ao software. Nesta licença ainda é excluída explicitamente a responsabilidade de não infração, que pode ocorrer, por exemplo, quando alguém usa a propriedade intelectual de outra pessoa sem a devida autorização. Isso cobre casos em que o autor do software, acidentalmente ou não, tenha utilizado algum material protegido sob direitos autorais ou uma ideia patenteada sem obter uma licença para tanto, o que pode gerar um processo contra o autor, os distribuidores do software e até seus usuários. Porém, assim como explicado anteriormente, a ausência de responsabilidades tem sua validade limitada pelas leis vigentes.

Vantagens e Desvantagens.

Essa licença é a recomendada pela *Free Software Foundation* quando se busca uma licença permissiva, pois é bastante conhecida e, ao contrário da BSD, não possui múltiplas versões com cláusulas que podem gerar dificuldades adicionais, tais como a cláusula de propaganda da BSD que pode gerar incompatibilidades com outras licenças. Outra vantagem da MIT em relação à BSD é a maior clareza dos seus termos ao declarar explicitamente que é permitido, por exemplo, sublicenciar ou vender cópias do software, que são direitos apenas implícitos na BSD. A questão do sublicenciamento é bastante importante quando o software será usado como parte de um trabalho coletivo ou derivado que será distribuído sob outra licença, o que é bastante comum nas práticas de software livre. Se não houver o direito de sublicenciamento, então apenas o detendor do direito autoral pode conceder a licença. Desta forma, o usuário de um trabalho derivado precisa obter a licença tanto do autor desse trabalho como também dos detentores dos direitos de cada componente que faz parte dele, sendo necessário identificar todos esses componentes e pessoas envolvidas, aumentando a dificuldade, os riscos e a complexidade jurídicica. Por outro lado, quando é permitido o sublicenciamento, a pessoa que está emitindo a licença, que valerá para todos os componentes de seu software, fica responsável por analisar os termos de cada um dos componentes e certificar-se da compatibilidade entre as licenças. Assim, ao chegar no usuário final, o potencial de problemas é restrito a uma única licença.

4.1.3. A Licença Apache

A Licença Apache está atualmente na versão 2.0 (**www.apache.org/licenses/LICENSE-2.0.html**) e é usada por um dos projetos mais conhecidos de software livre, o servidor Web Apache, assim como pela maior parte dos outros projetos pertencentes à Fundação Apache, além de projetos independentes que optaram por usar essa licença.

A Apache também é uma licença permissiva e, na versão 1.1, seu texto era bastante similar ao da BSD, porém dando ênfase à proteção da marca Apache. Havia uma cláusula similar à cláusula de propaganda da BSD, obrigando que a documentação ou o software, quando redistribuídos, incluíssem a frase “este produto inclui software desenvolvido pela Apache

Software Foundation (http://www.apache.org/)" e duas cláusulas proibindo o uso do nome Apache sem permissão escrita prévia, tanto para endossar trabalhos derivados, como na BSD, como também o uso como parte do nome do produto. Em 2004, a licença foi totalmente reescrita e seu texto ficou bem mais longo e complexo, detalhando melhor os direitos concedidos, conforme será visto a seguir.

A primeira parte da licença contém as definições de palavras-chave que serão utilizadas no decorrer da licença, tais como "Entidade Legal", "Você" e "Fonte". A definição de fonte é mais abrangente do que a convencional, englobando não apenas o código fonte do software, como também fonte da documentação e arquivos de configuração. Analogamente, na definição de "Objeto" é incluída qualquer coisa resultante da transformação ou tradução mecânica da fonte, tais como código compilado, documentação e conversões para outros tipos de mídia. Essa licença também define o que é um "Trabalho Derivado", excluindo explicitamente trabalhos que se mantém separados do trabalho sendo licenciado ou que meramente são ligados à sua interface. A próxima definição é a de "Contribuição", que é uma modificação do trabalho original que será incluída no trabalho em futuras versões. Como será visto adiante, a contribuição também pode ser licenciada nos termos da licença Apache.

As próximas duas cláusulas se referem à concessão de direitos autorais e patentes. Quanto aos direitos autorais, é dada uma licença irrevogável para reproduzir, preparar trabalhos derivados, mostrar publicamente, sublicenciar e distribuir o trabalho e seus derivados, na forma de fonte ou objeto, sujeitos aos termos e condições da licença. Essa cláusula enumera os direitos protegidos pela lei que estão sendo concedidos à pessoa que obtém a licença, adaptando-se bem às leis americanas, porém no caso brasileiro, em que a legislação de direitos autorais é muito mais restritiva, acaba deixando de lado alguns direitos fundamentais, como por exemplo o do uso do software. Nesse sentido, a versão anterior da licença era mais apropriada, já que definia os direitos concedidos de forma mais abrangente e incluia em seu texto a permissão de uso.

Em seguida, são colocadas algumas condições para a redistribuição do trabalho e seus derivados:

- incluir uma cópia da licença;
- incluir avisos em todos os arquivos modificados informando sobre a alteração;

- manter na fonte de trabalhos derivados todos os avisos de direitos autorais, patentes e marcas registradas que são pertinentes;
- se o trabalho incluir um arquivo texto chamado “NOTICE”, então qualquer trabalho derivado distribuído deve incluir os avisos pertinentes contidos nesse arquivo da forma como está detalhado na licença.

A cláusula sobre redistribuição termina informando explicitamente que é permitido licenciar sob outros termos qualquer modificação ou trabalho derivado, desde que o uso, reprodução e distribuição do trabalho que foi obtido pela licença Apache esteja de acordo com seus termos.

Já no caso das contribuições enviadas, que são tratadas na cláusula seguinte, fica determinado que é usada a princípio a licença Apache, sem termos ou condições adicionais, a não ser que tenha sido realizado algum outro tipo de acordo de licenciamento entre as partes.

As quatro últimas cláusulas tratam das marcas registradas e responsabilidades legais. A licença limita-se a permitir o uso da marca apenas para uso razoável para descrever a origem do trabalho e reproduzir o arquivo NOTICE. Dessa forma, fica implícito que não é permitido associar os nomes relacionados ao trabalho original a trabalhos derivados além do necessário para indicar a origem do trabalho. A cláusula que informa sobre a falta de garantias e responsabilidade é bastante similar àquela vista anteriormente na licença MIT, assim como a cláusula sobre limitações da responsabilidade. A principal diferença está na proteção das contribuições e seus responsáveis, já que esses também são tratados no resto da licença. Na última cláusula, é tratada a questão de outras entidades aceitarem prover garantias ou serem responsabilizadas pelo trabalho, o que é permitido desde que fique claro que a entidade está agindo por conta própria e ficará totalmente responsável pelas ações judiciais decorrentes, de forma a não interferir nas proteções judiciais que recaem sobre os contribuidores de acordo com as cláusulas anteriores da licença.

Após apresentar os termos acima, a licença inclui instruções sobre como aplicá-la a um trabalho. Como o texto da licença é bastante extenso, tornando inconveniente apresentá-lo por completo em cada arquivo do código fonte do projeto, a licença dá como diretriz que seja usado o seguinte texto:

Copyright [ano] [nome do detentor dos direitos]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

Vantagens e Desvantagens.

A principal vantagem da licença Apache é seus termos estarem definidos de forma mais precisa, deixando menos margem a interpretações conflitantes com os interesses dos envolvidos. Em particular, o fato da licença Apache deixar explícito na cláusula 4, sobre redistribuição, que é permitido o uso de outra licença, é visto como uma vantagem sobre as licenças BSD ou MIT quanto à clareza de sua característica permissiva. Outra vantagem dessa licença é o tratamento de contribuições, tornando mais transparente a incorporação das mesmas em um projeto quando é feita a opção por manter a mesma forma de licenciamento. Para formalizar o processo de contribuições, foi criado o *Apache Contributor License Agreement*, que transmite à *Apache Software Foundation* todos os direitos de propriedade intelectual necessários para que a fundação possa licenciar a contribuição, o que é importante caso ocorra a decisão de uma alteração de licença do projeto. O fato do texto da licença ser mais extenso do que os vistos anteriormente é visto como uma desvantagem por algumas pessoas, porém o ganho em precisão compensa esse fato. O principal problema da Apache é o fato de sua compatibilidade com a GPL 2.0 ser discutível, dado que ela impõe algumas restrições que não estão no texto dessa versão da GPL.

4.2. Recíprocas Totais

As licenças recíprocas totais determinam que qualquer trabalho derivado precisa ser distribuído sob os mesmos termos da licença original. Isso também é chamado de *copyleft*, um termo criado pela *Free Software Foundation* (www.gnu.org/copyleft). A ideia do *copyleft* é dar permissão a todos para executar, copiar, modificar e distribuir versões modificadas do programa, mas impedir que sejam adicionadas restrições a essas versões redistribuídas. Tal ideia visa fortalecer o software livre como um todo, não permitindo que melhorias do software sejam retiradas do alcance da comunidade. O resultado esperado é que a quantidade de software livre aumente cada vez mais, beneficiando todos os envolvidos na cadeia produtiva do software livre. Além disso, a reciprocidade contribui para manter a compatibilidade entre diversas versões de um determinado sistema, dado que quando novas funcionalidades são feitas de forma fechada, fica mais difícil replicá-las nas diferentes versões. Por outro lado, tal abordagem também sofre críticas de dentro da comunidade, pois o software licenciado nesse modelo acaba ficando de certa forma isolado dos demais devido a incompatibilidades nas licenças. Na prática, software licenciado sob o modelo permissivo, em geral, pode ser incorporado em software licenciado como recíproco, já que licenças permissivas permitem a redistribuição sob outros termos, inclusive os de licenças recíprocas. Porém, o inverso não é verdadeiro e, assim, software sob licenças recíprocas não pode ser utilizado em vários projetos de software livre que usam alguma outra licença.

A licença que deu origem à ideia de *copyleft* foi a *General Public License*, ou GPL (**www.gnu.org/licenses/gpl.html**), da *Free Software Foundation*. Nesta seção iremos estudar algumas de suas versões: a GPL 2.0, GPLv3 e *Affero General Public License*, ou AGPL.

4.2.1. GPL 2.0

A licença GPL (http://www.gnu.org/licenses/old-licenses/gpl-1.0.html) foi escrita em 1989 pela *Free Software Foundation*. Dois anos depois, em junho de 1991, foram feitas pequenas modificações na licença, gerando a versão 2.0 (http://www.gnu.org/licenses/old-licenses/gpl-2.0.html). Essa versão manteve-se até 2007, quando saiu a GPLv3, que será vista na próxima seção. Devido ao grande número de projetos licenciados sob a versão 2.0, incluiremos neste relatório uma descrição detalhada dessa versão da licença.

A GPL 2.0 inclui um preâmbulo que explica os princípios que baseiam a licença e seus principais objetivos. Apesar desse preâmbulo ser bastante citado nas discussões, ele não tem valor jurídico, ou seja, por não fazer parte dos termos e condições, suas palavras não precisam ser obedecidas por quem obtém a licença do software. Seu objetivo é apenas melhorar o entendimento da GPL em seu contexto [Ros05], explicando o que é software livre e a importância do *copyleft*.

A licença GPL pode ser copiada, distribuída e aplicada a qualquer software cujo detentor dos direitos autorais assim desejar. Porém, diferentemente de outras licenças, como a BSD, o texto da GPL não pode ser alterado sem autorização, ou seja, não é permitido que seja feita uma licença derivada dela. No final da licença, há uma explicação sobre como aplicá-la a um trabalho. A *Free Software Foundation* recomenda que o autor que usa a GPL permita que seu trabalho esteja licenciado sob a versão mais recente da licença ou qualquer versão posterior, de forma que quando surgir uma nova versão o usuário da licença possa escolher qual das versões estará utilizando. Dessa forma, evita-se incompatibilidade entre programas mais antigos e mais novos que optaram por utilizar a GPL. Porém, muitas pessoas preferem ter maior controle sobre quais são os termos em que seu software está licenciado e, assim, não deixam aberto para qualquer versão posterior criada pela *Free Software Foundation*. Um exemplo importante dessa escolha é o núcleo do Linux.

A Seção 0 da GPL define ao que ela se aplica, que seria qualquer programa ou trabalho que contém uma nota afirmando que está sendo licenciado pelos termos da GPL. No resto da licença, esse objeto a ser licenciado é chamado simplesmente de *Programa*. A Seção 0 também afirma que a licença se aplica a qualquer trabalho derivado segundo a lei de direitos autorais, porém ao mesmo tempo define trabalho derivado como "um trabalho contendo o Programa ou parte dele, literalmente ou com modificações e/ou traduzido para outra língua" (GNU General Public License, version 2, 1991). Essa definição dada a trabalho derivado difere tanto da legislação americana quanto da brasileira. Segundo a definição dada pela GPL, mesmo trabalhos coletivos são considerados como trabalhos derivados. Tal definição é de importância fundamental na aplicação da GPL e motivo de muita controvérsia quanto à necessidade de reciprocidade em diferentes usos do software. Além disso, a Seção 0 afirma que o ato de executar o Programa não é restrito de nenhuma forma e que a

saída do Programa só está coberta pela licença se seu conteúdo constituir de alguma forma um trabalho baseado no Programa, o que só aconteceria em casos muito específicos como, por exemplo, um programa que gera outro programa que é uma versão modificada de si mesmo.

A próxima seção trata da cópia e distribuição do código fonte do Programa, que pode ser realizada desde que se mantenha os avisos sobre o *copyright*, a ausência de garantias e a licença. Nesta seção também é explicado que é permitido exigir pagamento pelo ato de transferir uma cópia ou por garantias adicionais que a pessoa decida oferecer, o que permite o uso do software em um modelo de negócio comercial.

A Seção 2 da GPL é uma das mais importantes da licença, pois é onde estão definidas as regras relacionadas ao conceito de *copyleft*, tratando das modificações do programa e sua distribuição. As condições impostas, além daquelas que foram estipuladas para a distribuição segundo o parágrafo anterior, são as seguintes:

1. "Os arquivos modificados precisam conter avisos proeminentes afirmando que os arquivos foram modificados e a data da modificação", de forma a proteger a reputação do autor do trabalho original, que não fica sujeito a problemas causados pela versão modificada.
2. "Qualquer trabalho distribuído ou publicado que contém totalmente ou em parte o programa ou que é derivado do mesmo ou parte dele precisa ser licenciado como um todo sem nenhuma cobrança sob os termos desta licença", o que significa que qualquer trabalho derivado está sujeito às mesmas restrições da GPL. Ou seja, não é possível que um trabalho derivado de um programa GPL seja licenciado de modo fechado, permissivo, ou qualquer outra licença, a não ser que haja a opção de utilizar outra licença para esse programa.
3. "Se o programa modificado normalmente lê comandos de forma interativa quando é executado, é necessário que, quando ele começar a ser executado normalmente, ele imprima ou mostre um aviso incluindo as informações sobre *copyright* e ausência de garantias (ou que uma garantia é provida pela pessoa) e que os usuários podem redistribuir o programa sob essas condições e informando o usuário como ver uma cópia dessa licença". É feita uma exceção: tal regra não precisa ser seguida no caso do programa original ser interativo mas não mostrar tal aviso. O objetivo deste item é

garantir que as pessoas sejam informadas sobre seus direitos, de forma que possam utilizá-los da forma como lhes for mais interessante.

Em seguida, a Seção 2 define, com maiores detalhes, as condições em que são aplicadas as regras acima. Porém, as explicações dadas ainda estão longe de esclarecer de fato o impacto da reciprocidade em todos os casos de combinação de um programa GPL com o de outra licença. É explicado que as condições acima aplicam-se ao trabalho modificado como um todo. Os termos da GPL não precisam ser aplicados a seções que possam ser consideradas independentes e não derivadas do programa, desde que tais seções sejam distribuídas como trabalhos separados. Por outro lado, quando essas mesmas seções são distribuídas como parte de um todo que é um trabalho baseado no programa, então a distribuição desse todo precisa estar nos termos da GPL, cujas permissões se estendem para cada uma de suas partes independentemente de quem a escreveu. Isso implica que, para que o trabalho possa ser distribuído, cada uma de suas partes precisa no mínimo ter uma licença que seja compatível com os termos da GPL, já que no trabalho como um todo tais condições serão aplicadas.

A Seção 10 da GPL, que será vista adiante, provê algumas instruções sobre o que fazer quando as licenças não são compatíveis. Para evitar tal problema, alguns projetos optam por distribuir as partes necessárias para seu funcionamento separadamente, instruindo os usuários sobre como obter as partes que não estão sob seu controle e que poderiam gerar problemas de licença. É explicado que a intenção da Seção 2 não é reivindicar direitos de trabalhos criados inteiramente por outra pessoa, mas sim exercer o direito de controlar a distribuição de trabalhos derivados ou coletivos baseados no Programa. A seção finaliza afirmando que a mera agregação do Programa com outro trabalho que não seja baseado no Programa em um volume de armazenamento ou mídia de distribuição não faz com que o outro trabalho caia no escopo desta licença (GNU General Public License, version 2, 1991). Isso evita interpretações que exagerem nas consequências da licença, limitando seu alcance a trabalhos que realmente são de alguma forma derivados do programa licenciado sob a GPL.

Na próxima seção, é determinado que para distribuir o Programa é necessário que ele esteja acompanhado do código fonte. Como alternativa a distribuir o código junto com o Programa, é permitido que seja feita uma oferta por escrito, com validade mínima de três anos, de que o código seja

enviado mediante uma cobrança não superior ao custo de fazer tal distribuição física. Uma pessoa que recebeu tal oferta pode reutilizá-la em sua própria redistribuição do código, desde que seja para fins não-comerciais. Em geral, a não ser que o código seja muito grande, quem distribui programas GPL já disponibiliza o código junto com o binário, de forma a evitar trabalho e custos adicionais tanto para quem está distribuindo como para quem está recebendo. Essa seção define, ainda, que o código fonte inclui todos os módulos que ele contém, arquivos de definição de interface associados e *scripts* usados para controlar a compilação e instalação do executável. Porém, não é necessário incluir partes que normalmente são distribuídas junto com o sistema operacional em que o executável será usado, como por exemplo bibliotecas que já são parte da linguagem de programação utilizada. Essa seção finaliza explicando que se a distribuição do executável é realizada oferecendo acesso para que ele seja copiado de um determinado lugar, então disponibilizar o código fonte no mesmo lugar conta como distribuição do código fonte.

As Seções 4 a 6 da GPL continuam a explicação sobre o funcionamento da licença. Na seção 4 observa-se que, diferentemente de outras licenças que baseiam-se principalmente nas leis de contrato, na GPL há um foco também nas leis de direitos autorais. É ressaltado o fato de que somente a licença permite que uma pessoa modifique ou distribua o Programa ou um derivado dele, já que tais atos são protegidos pelas leis de direito autoral. Se a pessoa não seguir os termos da licença, ela perde esses direitos. Porém, isso não interfere nas pessoas que receberam os direitos a partir daquela que infringiu a licença. Essas pessoas continuam usufruindo dos diretos desde que elas próprias não cometam nenhuma infração. Isso acontece porque, conforme explicado na Seção 6, quando o Programa é redistribuído, o recipiente automaticamente recebe uma licença do detentor original dos direitos, e não do seu intermediário. Essa relação criada diretamente entre quem emite a licença e quem a recebe permite também que o detentor dos direitos entre com um processo contra qualquer infrator da licença. A Seção 6 também afirma que "Você não poderá impor aos recebedores qualquer outra restrição ao exercício dos direitos então adquiridos". Na prática, essa limitação causa a incompatibilidade de outras licenças com a GPL. Devido a isso e às regras impostas na Seção 2, software distribuído sob licenças que têm alguma restrição não pode ser combinado com software GPL.

As Seções 7 e 8 da GPL continuam afirmando a necessidade de seguir os termos da licença, mesmo na presença de outros fatores geradores de

outras obrigações, como, por exemplo, decisões judiciais e leis locais. Caso não seja possível cumprir tais obrigações e ao mesmo tempo seguir os termos da licença, então não é permitido que o Programa seja distribuído. Essa parte da licença é de particular importância em países onde há patentes de software, pois no caso de uma decisão judicial tentar incluir restrições relacionadas a um software, ele não pode mais ser distribuído sob a GPL. Dessa forma, o modelo de software livre proposto pela GPL é garantido mesmo na presença de decisões judiciais que iriam contra seus princípios. É permitido ao detentor dos direitos autorais limitar a distribuição do seu Programa onde as leis locais apresentam conflitos com a GPL.

Pra finalizar, as duas seções seguintes da GPL provêem mais duas explicações sobre a licença. A Seção 9 trata da política de versões das licenças publicadas pela *Free Sofrware Foundation* e as possibilidades de escolha da versão da licença conforme o modo como a licença é especificada no Programa. Já a Seção 10 recomenda que, se uma pessoa quer incorporar o Programa em algum outro software livre cujas condições de distribuição sejam diferentes, então ela deve entrar em contato com o autor para tentar conseguir uma permissão específica. A licença termina de forma similar às licenças vistas anteriormente, com duas seções declarando a ausência de garantias. Após os termos e condições, a licença inclui, ainda, alguns parágrafos sobre como aplicá-la ao programa a ser licenciado.

A *Free Software Foundation* informa que bibliotecas que poderão ser incorporadas em software fechado utilize a licença LGPL, que será vista adiante. Segundo ela, a GPL não permite que um programa coberto por essa licença seja ligado a software licenciado sob outros termos, conforme explicado na Seção 2. Tal interpretação é causa de controvérsia entre os estudiosos de licenças de software livre. Lawrence Rosen, em seu livro *Open Source Licensing*, uma das principais fontes sobre o assunto, argumenta que o uso de uma biblioteca cria um trabalho coletivo e não um trabalho derivado. Portanto, não seria necessário licenciar o software como GPL quando distribuído separadamente. Como essa interpretação é contrária à da *Free Software Foundation*, autora da GPL, Rosen afirma ainda que o que importa é o entendimento entre o detentor dos direitos e a pessoa que recebe a licença. Ele cita como exemplo o caso do núcleo do Linux, em que seu autor, Linus Torvalds, apesar de usar a GPL, adotou a política de que software que é apenas combinado com o Linux não está sujeito aos termos da GPL, independentemente de como esse software é ligado ou distribuído [Ros05].

Dessa forma é possível a existência de *drivers* fechados no Linux. Porém, considerando a legislação brasileira, essa interpretação de trabalho coletivo não é cabível.

Vantagens e Desvantagens.

A GPL é bastante conhecida, sendo a licença mais utilizada em projetos de software livre. Porém, é considerada uma licença de alta complexidade. Apesar da intenção da licença estar clara em seu preâmbulo, há vários detalhes presentes em seus termos que dificultam sua interpretação em casos específicos.

A GPL é recomendada para projetos que buscam seu crescimento através de contribuições de terceiros, dado que melhorias feitas ao software devem manter-se livres para poderem ser distribuídas. A GPL também é usada frequentemente em um modelo comercial de licenciamento duplo. Neste caso, a empresa provê o software sob a licença GPL, obtendo os benefícios relacionados ao software livre, mas ao mesmo tempo disponibiliza o software sob alguma outra licença que não imponha as restrições presentes na GPL. Dessa forma, empresas que têm interesse em usar o software de forma fechada podem obter uma licença alternativa, normalmente pagando um determinado valor para a empresa detentora dos direitos sobre o software.

Por fim, antes de adotar a GPL é muito importante verificar sua compatibilidade com licenças de outros programas que serão utilizados no projeto, de forma a evitar ter que reescrever partes do software que já estariam disponíveis sob alguma outra licença.

4.2.2. GPLv3

A mais nova versão da GPL (www.gnu.org/licenses/gpl-3.0-standalone.html), lançada em 29 de junho de 2007, após um longo período de discussão e revisão pública, foi criada para evitar algumas situações consideradas indesejáveis pela *Free Software Foundation*. Além disso, algumas partes foram reescritas de forma a adaptar a licença a novas formas de compartilhamento de programa e a deixá-la mais adequada para legislações em que os termos originais poderiam ser interpretados de forma diferente da esperada pela *Free Software Foundation*. Também foram realizadas alterações para facilitar a compatibilidade com outras

licenças, em particular a Apache 2.0. A seguir serão discutidas as principais mudanças.

Um dos principais motivos para mudar para a GPLv3, segundo seus criadores, é evitar o fenômeno conhecido como *tivoização* (em referência ao aparelho TiVo, que funciona como um gravador digital de vídeos). O TiVo inclui software derivado do Linux, licenciado sob a GPL 2.0. O código está disponível e pode ser modificado, porém, tais modificações não podem ser utilizadas no aparelho TiVo, pois ele faz uma checagem da assinatura digital do software e executa apenas as versões permitidas pelo fabricante. Essa questão de assinaturas digitais foi bastante controversa durante a elaboração da GPLv3, pois ao mesmo tempo em que as assinaturas restringem a liberdade dos usuários, defendida pela *Free Software Foundation*, elas são uma ferramenta importante para implementar segurança em alguns sistemas. A estratégia para impedir a *tivoização* é exigir que o fabricante provenha toda informação necessária para instalar versões modificadas do software no aparelho. Essa informação pode ir desde instruções básicas até chaves de autorização que possam ser necessárias. Porém, tal exigência é limitada a aparelhos considerados "produtos de usuário". Assim, alguns equipamentos de uso específico, como por exemplo máquinas de votação, estariam isentos da necessidade de reduzir seu nível de segurança para se adequar a licença. A definição de "produtos de usuário" está presente na Seção 6 da licença, sobre distribuição na forma binária:

> A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

Outra mudança na GPLv3 é relacionada a mecanismos de DRM, ou *Digital Rights Management* (www.defectivebydesign.org). É sabido que a

Free Software Foundation é contra o uso de DRM, porém, não quiseram impedir que software livre fosse utilizado para implementá-lo, já que isso estaria limitando a liberdade dos usuários. Como alternativa, decidiram atacar o tratado de *copyright* da *World Intellectual Property Organization* (WIPO), adotado em 20 de dezembro de 1996. Nesse tratado, o artigo 11, sobre obrigações a respeito de medidas tecnológicas, afirma que os países devem adotar medidas legais para reprimir tentativas de burlar sistemas tecnológicos de proteção usados por autores para restringir atos que não são autorizados, em conexão com o exercício de seus direitos de acordo com esse tratado ou a Convenção de Berna. A solução presente na GPLv3 é afirmar que qualquer trabalho sob a GPL não pode ser considerado uma "medida tecnológica efetiva". Ou seja, é permitido incluir sistemas de DRM no software tanto quanto é permitido quebrá-lo. Tal cláusula está na seção três, citada abaixo:

> 3. Protecting Users' Legal Rights From Anti-Circumvention Law.
>
> No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.
>
> When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

Outro ponto tratado em maior profundidade na nova versão da GPL é a questão das patentes. Uma das motivações para as mudanças que foram implementadas na licença foi um acordo entre a Microsoft e a Novell, decorrente de uma alegação da Microsoft de que a Novell estaria infringindo suas patentes na distribuição SUSE Linux. Segundo o acordo, a Microsoft não processaria usuários por infração de patentes desde que o software fosse obtido de alguém

que estivesse pagando à Microsoft para obter tais direitos. A GPLv3 é bem específica quanto ao caso e, na Seção 11, sobre patentes, afirma que uma organização não poderá distribuir trabalhos cobertos por essa licença caso faça parte de um desses acordos discriminatórios, conforme detalhado a seguir:

> A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Há ainda outras definições sobre patentes nessa seção, porém, elas não serão detalhadas aqui, já que não são relevantes dentro da legislação brasileira. Mas essa questão das patentes é uma das principais causas de receio por parte das empresas que detêm propriedade intelectual nessa forma. Como a comunidade de software livre muitas vezes busca o apoio dessas empresas, esse tornou-se um forte fator para frear a adoção da GPLv3.

Quanto à compatibilidade com outras licenças, a mais importante delas sendo a Apache, foram adicionados alguns termos na Seção 7 da GPL, intitulada "Termos Adicionais", de forma a permitir que software cuja licença possua certas restrições ainda assim possa ser relicenciado sob a GPL, e dessa forma ser distribuído como parte de um trabalho derivado ou coletivo:

> Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:
>
> * a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

> * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
>
> * c) Prohibiting misrepresentation of the origin of that material,or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
>
> * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
>
> * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
>
> * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

Para a distribuição de código binário, foram adicionadas novas possibilidades de disponibilização do código fonte correspondente, a mais importante delas tratando do compartilhamento *peer-to-peer*. Nesse caso, basta que seja informado onde o código fonte está sendo oferecido ao público. Além disso, segundo a Seção 9, a mera transmissão do programa através de *peer-to-peer* não obriga que o usuário aceite a licença. Ou seja, apenas a pessoa que inicia o processo de distribuição *peer-to-peer* precisa se preocupar em informar sobre o código fonte, as demais estando isentas de qualquer obrigação.

Por fim, vale comentar que há algumas mudanças sutis na forma como a licença foi escrita, visando evitar ambiguidade de seus termos. Por exemplo, a palavra *distribute* foi substituída por *convey*, termo que foi definido no início da licença como “qualquer forma de propagação que permite a outras partes fazer ou receber cópias”.

Vantagens e Desvantagens.

Considerando que essa é apenas uma atualização da GPL, as vantagens e desvantagens analisadas aqui serão em relação a sua versão anterior, que já foi discutida. A GPLv3 foi lançada para corrigir algumas brechas que foram verificados no decorrer dos anos com o uso da GPL 2.0, deixando a licença mais alinhada à visão de *copyleft* da *Free Software Foundation*.

Seu texto passou por um longo período de revisão aberta pela comunidade, em que foi possível sugerir e incorporar diversas melhorias. Sendo assim, é um texto bastante sólido e bem escrito, que está consistente com uma ampla gama de legislações, inclusive a brasileira. A revisão da GPL também serviu para alinhá-la às atuais práticas de distribuição de software, incluindo o compartilhamento em redes *peer-to-peer*.

Uma das principais vantagens da nova versão da GPL é sua compatibilidade com um maior número de licenças de software livre, principalmente a Apache, que é uma das licenças permissivas mais usadas pela comunidade. Porém, a GPLv3 também apresenta uma forte desvantagem em relação a compatibilidade: software que está sob a licença GPL 2.0 (sem a cláusula "ou posterior") não pode ser integrado a software GPLv3, pois essas licenças são incompatíveis.

Outra desvantagem é que várias empresas têm um certo receio em adotar a GPLv3, pois ela é uma licença relativamente nova e de alta complexidade. A cláusula que gera maior preocupação nos advogados corporativos é a relacionada às patentes.

Portanto, se o objetivo é garantir as liberdades propostas pelo movimento de software livre, a GPLv3 é mais adequada do que sua anterior. Porém, se uma maior difusão do software é o mais importante, tornando-o compatível com um maior número de licenças e incentivando seu uso em qualquer empresa, então deixar sob a licença "GPL versão 2 ou superior" pode ser uma melhor alternativa. Note que se o projeto envolver componentes cuja licença é compatível apenas com a GPLv3, então é necessário que a licença adotada seja a GPLv3, e não "GPL versão 2 ou superior".

4.2.3. AGPL

A Affero Inc. (**www.affero.org**) é uma empresa que se define com a missão de "trazer a cultura de patrocício para a Internet". Ela provê um serviço

de hospedagem de páginas pessoais para autores de diversos tipos e integra um sistema de pagamento seguro para que pessoas possam fazer doações. A Affero apóia o desenvolvimento de software livre e, em março de 2002, criou a primeira versão da *Affero General Public License*, ou AGPL (**www.affero.org/oagpl.html**). A AGPL é uma adaptação da GPL, autorizada pela *Free Software Foundation*, que inclui um termo sobre uso de um software através de uma rede. O parágrafo adicionado à seção 2, sobre modificação e cópia do programa e sua distribuição, é o seguinte:

> d) If the Program as you received it is intended to interact with users through a computer network and if, in the version you received, any user interacting with the Program was given the opportunity to request transmission to that user of the Program's complete source code, you must not remove that facility from your modified version of the Program or work based on the Program, and must offer an equivalent opportunity for all users interacting with your Program through a computer network to request immediate transmission by HTTP of the complete source code of your modified version or other derivative work.

Esse parágrafo afirma, de forma resumida, que se no programa original os usuários que interagiam com o programa tinham a opção de pedir o código fonte completo, tal opção tem que ser mantida em qualquer versão modificada. Dessa forma, mesmo não havendo a distribuição de um binário, um aplicativo web público sob esta licença precisa se manter aberto para qualquer usuário que interaja com ele.

Em 19 de novembro de 2007, a *Free Software Foundation* lançou a *GNU Affero General Public License*, conhecida como AGPLv3 (www.fsf.org/licensing/licenses/agpl-3.0.html), uma licença diferente daquela escrita pela Affero e compatível com a GPLv3. Para permitir que um programa licenciado sob a antiga licença Affero que utilizasse a cláusula "ou posterior" pudesse ser convertido para essa nova licença, a Affero Inc. criou uma versão intermediária da licença, que determina apenas o seguinte:

> This is version 2 of the Affero General Public License. It gives each licensee permission to distribute the Program or a work based on the Program (as defined in version 1 of the Affero GPL) under the GNU Affero General Public License, version 3 or any later version.

> If the Program was licensed under version 1 of the Affero GPL "or any later version", no additional obligations are imposed on any author or copyright holder of the Program as a result of a licensee's choice to follow this version 2 of the Affero GPL.

As diferenças entre a GPLv3 e a AGPLv3 residem no preâmbulo e na seção 13 da licença. Enquanto na GPLv3 a seção 13 informa apenas que trabalhos que usam essas duas licenças podem ser combinados, valendo a AGPLv3 como licença do trabalho resultante, a seção 13 da AGPLv3 afirma ainda que:

> Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Essa nova versão da redação da cláusula visa tornar a Affero GPL ainda mais abrangente. Enquanto originalmente era colocado como critério que o programa fosse feito para interagir com usuários através de uma rede, nessa nova versão, qualquer tipo de software que interage através da rede está coberto, como por exemplo servidores de jogos, que interagem com o usuário através de um outro programa intermediário.

Vantagens e Desvantagens.

Para a AGPL valem todas as considerações feitas a respeito da GPL. Ela é recomendada para projetos em que há interação via rede e busca-se o *copyleft*. A AGPL é considerada a mais "viral" das licenças, portanto deve ser evitada em projetos em que haja qualquer expectativa de utilização

sob outra licença, a não ser que seja adotado um modelo de múltiplo licenciamento.

4.3. Recíprocas Parciais

Licenças recíprocas parciais, também chamadas de *copyleft fraco*, determinam que modificações do trabalho coberto por elas devem ser disponibilizadas sob a mesma licença. Porém, quando o trabalho é utilizado apenas como um componente de outro projeto, esse projeto não precisa estar sob a mesma licença. Alguns autores, como Simon Phipps [Sun06], utilizam a denominação licença baseada em arquivo para essa categoria, enquanto as recíprocas totais seriam licenças baseadas em projeto.

Considera-se que essas licenças são as que melhor equilibram dois importantes fatores do modelo de software livre: atração de interesse para a comunidade e força e longevidade do código-fonte disponível. Ao mesmo tempo que essas licenças permitem que os desenvolvedores utilizem o trabalho para criar software que será licenciado como preferirem, modificações e melhorias feitas ao próprio trabalho são obrigatoriamente disponibilizadas à comunidade [Sun06].

A *Free Software Foundation* recomenda o uso desse tipo de licença apenas em casos específicos. Seu argumento é a necessidade de fortalecer o software livre em detrimento do software fechado. Assim, quando as funcionalidades de uma biblioteca não estão facilmente disponíveis para uso em software fechado, seria melhor mantê-las dessa forma, utilizando uma licença recíproca total. Dessa forma, o software livre teria uma vantagem sobre concorrentes fechados. Porém, se as funcionalidades já estão ao alcance de software fechado, e portanto essa vantagem não está em questão, então uma licença recíproca parcial é recomendada, pois esse modelo ajuda a aumentar o número de usuários da biblioteca [Fre09].

Alguns advogados, como Lawrence Rosen [Ros05], defendem que o uso de bibliotecas que são apenas ligadas a um novo software não caracterizaria uma trabalho derivado, mas sim um trabalho coletivo. Ele faz uma analogia a páginas na web, em que cada uma é um trabalho com direito autoral individual, apesar de muitas vezes estarem presentes ligações de uma para a outra. Segundo ele, esse tipo de relação consiste em um trabalho coletivo. Portanto, nesse cenário, mesmo um software sob uma licença recíproca total poderia ser usado como biblioteca de outro que estaria sob outra licença. Porém, no

caso brasileiro, a Lei de Direito Autoral é mais específica e impõe maiores limitações. Dessa forma, o uso de licenças recíprocas parciais faz-se necessário em casos nos quais o autor quer garantir que o desenvolvimento da biblioteca seja feito no modelo de software livre mas ao mesmo tempo quer permitir seu uso em projetos que utilizam outras licenças.

4.3.1. A Licença LGPL

A *GNU Lesser General Public License*, ou LGPL (www.fsf.org/licensing/licenses/lgpl.html), originalmente denominada *GNU Library General Public License*, foi escrita em 1991 pela *Free Software Foundation*. Assim como a GPL e a AGPL, passou por grandes modificações no final de 2007 para adequar-se à versão 3 das licenças.

Em versões anteriores, a LGPL era uma cópia da GPL com algumas modificações relativas a bibliotecas, definidas na licença como "uma coleção de funções de software e/ou dados preparada para ser convenientemente ligada com programas aplicativos (que usam algumas dessas funções e dados) para formar executáveis." Veremos a seguir os termos da versão 2.1 dessa licença. Nessa parte do trabalho, adotaremos a versão 2.0 da GPL quando esta for mencionada.

A primeira limitação presente na LGPL que não constava na GPL é que trabalhos modificados da biblioteca precisam ser também bibliotecas, segundo o artigo *a* da seção 2. Já o artigo *d* da mesma seção busca maximizar a utilidade da biblioteca, desvinculando-a de aplicações específicas. Ele determina que:

> If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

Para permitir a distribuição de bibliotecas LGPL junto com software GPL, há também uma cláusula na LGPL que afirma que pode-se optar por aplicar os termos da GPL ao invés dessa licença (a LGPL) para uma determinada cópia da biblioteca. Porém, é dito também que tal mudança é

irreversível para aquela cópia, e assim a GPL é aplicada a todas as cópias subsequentes e trabalhos derivados feitos a partir dela. Além disso, caso seja feito um trabalho derivado da biblioteca que não resulta em uma biblioteca, relicenciá-lo para GPL é a única alternativa.

A Seção 5 da LGPL afirma que trabalhos que apenas usam a biblioteca, considerados isoladamente, não estão sujeitos aos termos da licença. Porém, também descreve em detalhes vários casos de uso da biblioteca em que é necessário estar atento às condições da licença. A Seção 5 está copiada na íntegra a seguir:

> 5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.
>
> However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.
>
> When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.
>
> If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)
>
> Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables

> containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

Apesar dessa tentativa de especificar o que seria um “trabalho que usa a biblioteca”, em que não há restrições quanto ao licenciamento, em oposição a um “trabalho baseado na biblioteca”, que estaria sujeito aos termos da LGPL, há muitos casos em que tal diferenciação não está clara. Além dos casos omissos, é possível mais de uma interpretação para alguns casos abordados nessa seção.

Há ainda a Seção 6, que faz uma exceção às anteriores e provê termos adicionais para trabalhos que incluem partes da biblioteca na versão que será distribuída. Essa seção é em parte equivalente à Seção 3 da GPL, e sua leitura é recomendada. Segundo os termos dessa seção, se a pessoa que está distribuindo um programa que usa a biblioteca não tem permissão para distribuir algum dos componentes necessários para execução do programa e esse componente não é parte integrante do sistema operacional, então a distribuição do programa na forma executável não é permitida nos termos da LGPL.

Na próxima seção, é descrito o caso de juntar uma biblioteca coberta pela LGPL com alguma outra biblioteca, sob outra licença, para formar uma única biblioteca. As regras impostas para tanto são as seguintes: em primeiro lugar, é necessário que os termos das licenças permitam que as bibliotecas sejam distribuídas como trabalhos separados. Além disso, a parte da biblioteca que está sob LGPL deve estar disponível separadamente sob os termos da LGPL, ou sendo distribuída junto com o conjunto, ou sendo colocado um aviso informando onde obtê-la.

As demais seções da LGPL correspondem às seções 4 a 12 da GPL, apenas com as alterações necessárias para tratar de uma “Biblioteca” ao invés de um “Programa”, conforme definido nas respectivas licenças.

A nova versão da licença, LGPLv3, apresenta-se como um conjunto de termos adicionais à GPLv3, já discutida na Seção 4.2.2. Isso implica que a licença LGPL foi totalmente reescrita em relação à versão 2.1, porém seus princípios foram mantidos. As diferenças entre a LGPL 2.1 e a GPL 2.0 correspondem, em grande parte, do ponto de vista semântico, às permissões adicionais que constituem a LGPLv3.

Há seis cláusulas na LGPLv3. A primeira constitui as "definições adicionais", que explica o que será entendido por "biblioteca", "aplicação", "trabalho combinado", "código fonte correspondente mínimo" e "código da aplicação correspondente".

Em seguida, é explicado como e em que circunstâncias uma pessoa pode distribuir um trabalho que está sob a LGPLv3 sem estar sujeito à seção 3 da GPLv3 (vide termos 1, 2, 3 e 4 da licença, que não serão citados aqui por se tratar de um texto muito extenso, porém cujos detalhes são de suma importância para o uso correto da LGPLv3):

Também são definidos os termos para criar um trabalho que constituído por bibliotecas combinadas:

> 5. Combined Libraries.
>
> You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:
>
> a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
>
> b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

E o último termo da LGPLv3 trate de versões revisadas da licença, informando que a *Free Software Foundation* poderá publicar versões revisadas e/ou novas da LGPL e explicando que se a pessoa recebeu o trabalho licenciado sob uma certa versão da licença "ou qualquer versão posterior", então ela poderá escolher quais das versões da licença ela irá seguir. Se não é especificada uma versão, então a pessoa pode escolher qualquer versão já publicada.

Vantagens e Desvantagens.

A LGPL é uma licença de alta complexidade, que requer uma observação bastante atenta dos seus termos para evitar seu descumprimento, que pode acarretar em uma ação judicial. Contextos de uso da biblioteca diferentes em geral requerem ações diferentes por parte da pessoa usando a biblioteca. Apesar de todos os detalhes presentes na licença, há ainda muita margem para interpretação de casos que não estão bem definidos. Além disso, o próprio relicenciamento de trabalhos derivados como LGPL é limitado, às vezes forçando o uso da GPL, como foi visto anteriormente.

Apesar da dificuldade em usar a LGPL, ela é uma licença amplamente adotada, pois combina características permissivas e recíprocas de forma balanceada, trazendo vantagens de ambos os modelos, conforme foi discutido na introdução desta Seção.

A escolha entre a versão 2.0 ou a LGPLv3 pode ser tomada com base nos mesmos argumentos aprensentados na Seção 4.2.2, sobre a GPLv3

4.3.2. A Licença Mozilla

A *Mozilla Public License*, ou MPL, foi escrita por uma das executivas da Netscape, Mitchell Baker, que tornou-se uma das principais responsáveis pelo projeto Mozilla, atuando como CEO da Fundação Mozilla por um longo período.

A licença Mozilla é considerada bem escrita e serviu como modelo para muitas das licenças de software livre comerciais que a seguiram [Ros05]. Ela une características de licenças recíprocas e de licenças permissivas, e portanto também é categorizada como uma licença recíproca parcial. Na licença Mozilla, a delimitação é bastante clara: o código coberto pela licença deve ser redistribuído pelos termos da licença Mozilla, porém esse código também pode ser utilizado em trabalhos ampliados, que podem estar sob outra licença.

A primeira seção da licença, como em um contrato convencional, trata das definições, que são bastante precisas. A lista completa de definições pode ser vista abaixo, porém algumas merecem atenção especial:

- "uso comercial" significa qualquer distribuição ou outra forma de deixar o software disponível, não se limitando ao uso por empresas;

- "contribuidor" recebe uma definição especial nessa licença, diferindo-o tanto do desenvolvedor inicial como também dos usuários comuns que estão usando o projeto;
- "executável" é definido de forma ampla, como qualquer coisa que não é o código fonte;
- "código fonte" é definido em mais detalhes do que encontramos nas licenças vistas anteriormente. São permitidos *patches* e também comprimir o arquivo, desde que o software para descompressão esteja largamente disponível gratuitamente.

1. Definitions

1.0.1. "Commercial Use" means distribution or otherwise making the Covered Code available to a third party.

1.1. "Contributor" means each entity that creates or contributes to the creation of Modifications.

1.2. "Contributor Version" means the combination of the Original Code, prior Modifications used by a Contributor, and the Modifications made by that particular Contributor.

1.3. "Covered Code" means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

1.4. "Electronic Distribution Mechanism" means a mechanism generally accepted in the software development community for the electronic transfer of data.

1.5. "Executable" means Covered Code in any form other than Source Code.

1.6. "Initial Developer" means the individual or entity identified as the Initial Developer in the Source Code notice required by Exhibit A.

1.7. “Larger Work” means a work which combines Covered Code or portions thereof with code not governed by the terms of this License.

1.8. “License” means this document.

1.8.1. “Licensable” means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

1.9. “Modifications” means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

A) Any addition to or deletion from the contents of a file containing Original Code or previous Modifications.

B) Any new file that contains any part of the Original Code or previous Modifications.

1.10. “Original Code” means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

1.10.1. “Patent Claims” means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.11. “Source Code” means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor’s choice. The Source Code can be in a compressed or

archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. "You" (or "Your") means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

A Seção 2 da MPL trata do licenciamento do código fonte, afirmando que o desenvolvedor inicial está concedendo uma licença não exclusiva, livre de *royalties* e válida em todo mundo. Essa licença libera a propriedade intelectual (exceto patentes ou marcas registradas) para o uso, modificação, reprodução, exibição, performance, sublicenciamento e distribuição do código original (ou porções dele) com ou sem modificações e como parte ou não de um trabalho ampliado. Para o caso das patentes também são concedidas algumas permissões, porém, diferentemente da LGPL, aplicam-se apenas ao código original e não a modificações feitas nele. Nessa seção também constam as permissões concedidas por pessoas que contribuem modificações ao código, que são bastante similares às vistas anteriormente, concedidas pelo desenvolvedor inicial.

A Seção 3 trata das obrigações no ato da distribuição, que são similares às da GPL. As principais exigências são que:

- o código coberto seja distribuído sob os termos da MPL;
- o código fonte esteja disponível;
- as modificações estejam explícitas;
- o aviso legal esteja presente no código fonte;
- na distribuição de formas executáveis, as condições anteriores sejam cumpridas para o código fonte correspondente.

A principal diferença em relação à GPL está na Seção 3.7, referente a trabalhos ampliados, que são criados combinando código coberto com outro código que não está sob a licença MPL em um único produto. Nesse caso, os requerimentos da MPL recaem apenas ao código coberto, e não ao trabalho como um todo, como seria na GPL.

A Seção 4 da MPL apresenta mais uma diferença em relação à GPL. Na impossibilidade de cumprir algum dos termos da licença por questões judiciais, basta que a situação seja explicada em um arquivo chamado LEGAL e que os demais termos sejam respeitados. Assim, o uso do código não é impedido, como aconteceria com a GPL. Além disso, na Seção 8, é dado um prazo de 30 dias para que seja corrigido qualquer descumprimento dos termos antes que a licença seja terminada.

A Seção 6 afirma que a Netscape tem o direito de alterar a licença a qualquer momento, e que a pessoa que está exercendo os direitos garantidos pela licença pode escolher se seguirá os termos em que o trabalho foi licenciado ou a nova versão. Isso significa que, por outro lado, o desenvolvedor é obrigado a aceitar as modificações de licença como termos válidos para seu projeto que foi licenciado anteriormente. A Netscape transferiu os direitos de alterar a licença para a Fundação Mozilla em 2003, apesar do texto da licença ainda não ter sido modificado para refletir essa alteração. A Seção 6 permite ainda que seja criada uma licença derivada da MPL, evitando assim que a Mozilla tenha controle sobre os termos de licenciamento do projeto, desde que fique claro na versão modificada que ela não está associada à Mozilla ou à Netscape. Além disso, a Seção 13 explica o licenciamento múltiplo, que permite ao usuário escolher qual licença, entre as fornecidas pelo desenvolvedor, ele irá adotar.

As Seções 7, 9, 10, 11 e 12 tratam de garantias, responsabilidade e outros aspectos jurídicos relacionados. Para finalizar, a MPL apresenta o seguinte quadro, para aplicação da licença:

EXHIBIT A -Mozilla Public License.

"The contents of this file are subject to the Mozilla Public License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at **http://www.mozilla.org/MPL/**

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either

express or implied. See the License for the specific language governing rights and limitations under the License.

The Original Code is _____.

The Initial Developer of the Original Code is _____.

Portions created by _____ are Copyright (C) _____. All Rights Reserved.

Contributor(s): _____.

Alternatively, the contents of this file may be used under the terms of the _____ license (the "[___] License"), in which case the provisions of [_____] License are applicable instead of those above. If you wish to allow use of your version of this file only under the terms of the [____] License and not to allow others to use your version of this file under the MPL, indicate your decision by deleting the provisions above and replace them with the notice and other provisions required by the [___] License. If you do not delete the provisions above, a recipient may use your version of this file under either the MPL or the [___] License."

[NOTE: The text of this Exhibit A may differ slightly from the text of the notices in the Source Code files of the Original Code. You should use the text of this Exhibit A rather than the text found in the Original Code Source Code for Your Modifications.]

Vantagens e Desvantagens.

A licença Mozilla encoraja trabalhos ampliados. Até mesmo alguns tipos de projetos que seriam considerados trabalhos derivados pela lei de *copyright* podem usar outra licença. Basta que sejam seguidos os termos da MPL, que

em linhas gerais requerem apenas que os arquivos que contém código do trabalho original estejam sob a licença MPL.

Devido à clareza de seus termos, baseados em definições precisas, a MPL é uma licença mais fácil de entender e aplicar do que a LGPL. Portanto, quando se busca uma licença que combine as vantagens do modelo recíproco com o modelo permissivo, a MPL é uma ótima alternativa.

Porém, uma desvantagem da MPL em relação à LGPL é a incompatibilidade com a GPL. Não é possível juntar dos projetos que estejam sob as licenças MPL e GPL, pois a MPL obriga que o código original mantenha-se como MPL e a GPL obriga que o trabalho como um todo e consequentemente cada uma de suas partes seja GPL.

5. Conclusão

As licenças de software livre muitas vezes são documentos de alta complexidade jurídica, sendo necessário um estudo cuidadoso de seus termos para avaliar o que é permitido ou não ser feito com um software. O presente capítulo teve como objetivo auxiliar pessoas envolvidas no processo de desenvolvimento de software livre a escolher uma licença para seus projetos, assim como ajudar usuários a entender as limitações a que estão sujeitos.

A categorização das licenças em três grupos (permissivas, recíprocas totais e recíprocas parciais) é bastante útil para entender as implicações que a licença adotada tem na forma como o software poderá ser usado para criar e distribuir trabalhos derivados. Licenças permissivas, como a BSD e a Apache, são recomendadas quando o objetivo é uma maior disseminação do software, permitindo que qualquer um utilize-o como desejar. Já licenças recíprocas totais, como a GPL, visam fortalecer a comunidade de software livre, usando meios para garantir que qualquer alteração do programa que seja distribuída esteja disponível para uso e adaptação da comunidade. Além disso, essas licenças possuem uma característica “viral”, em que programas dependentes daqueles licenciados nesse modelo também precisam usar a mesma licença quando distribuídos em conjunto. Por fim, licenças recíprocas parciais, como a LGPL, funcionam como um meio termo dos dois modelos anteriores. Nesse caso, o software é desenvolvido usando o modelo de reciprocidade, porém, dependendo das circunstâncias, é permitida a distribuição de outros programas que o utilizam sem a necessidade de aplicar a mesma licença ao conjunto.

Os diferentes modelos de licenciamento de software livre podem gerar incompatibilidades entre componentes. Essas questões sobre compatibilidade, assim como a base jurídica do licenciamento de software no Brasil, são objeto de estudo do Centro de Competência em Software Livre do IME-USP (ccsl.ime.usp.br) dentro do contexto do Projeto Qualipso (**www.qualipso.org**).

6. References

[CK08] Martin Campbell-Kelly. Historical reflections will the future of software be open source? *Communications of the ACM*, 51(10):21–23, 2008.

[Con08] Forrester Consulting. Open Source Paves the Way for the Next Generation of Enterprise IT, Nov 2008.

[DOS99] Chris DiBona, Sam Ockman, and Mark Stone, editors. *Open Sources*. O'Reilly, Sebastopol, 1999.

[FJL05] Joaquim Falcão, Tercio Sampaio Ferraz Junior, Ronaldo Lemos, Juliano Maranhão, Carlos Affonso Pereira de Sousa, and Eduardo Senna. Estudo sobre o software livre. Technical report, Escola de Direito da Fundação Getúlio Vargas, Rio de Janeiro, Março 2005.

[Fre09] Free Software Foundation – The GNU Project. http://www.gnu.org, 2009.

[Kon01] Fabio Kon. O software aberto e a questão social. Technical Report RT-MAC-2001-07, Departamento de Ciência da Computação, IME-USP, maio 2001. http://www.ime.usp.br/~kon/papers/RT-SoftwareAberto.pdf.

[Lau04] Andrew M. St. Laurent. *Understanding Open Source & Free Software Licensing*. O'Reilly, Sebastopol, 2004.

[Ope09] Open Source Initiative. http://www.opensource.org, 2009.

[Ray01] Eric S. Raymond. *The Cathedral & The Bazaar*. O'Reilly, 2001.

[Ros05] Lawrence Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, New Jersey, 2005.

[Sou09] SourceForge. http://sourceforge.net, 2009.

[Sun06] Sun Microsystems. Free and Open Source Licensing. White paper available at http://www.sun.com/software/opensource/whitepapers/Sun_Microsystems_OpenSource_Licensing.pdf, Dezembro 2006.

[Tau04] Cezar Taurion. *Software Livre: Potencialidades e Modelos de Negócio*. Brasport, Rio de Janeiro, 2004.

[Web04] Steven Weber. *The Success of Open Source*. Harvard University Press, Cambridge, 2004.

Plataform Strategies and the OW2 Consortium

Cedric Thomas
OW2 Consortium
March 2009

Introduction

A previous paper published about a year ago[1] described the OW2 business ecosystem strategy. The paper explained how a business ecosystem provides a framework for different, even competing, organizations to share a common goal and analysed what it takes to drive such a multi-faceted structure.

In this paper we intend to explain how our business ecosystem strategy is supported by a platform strategy.

Although the word platform is most often used to identify specific products or services, in this paper we choose to apply it to the OW2 Consortium; we demonstrate that the OW2 organization itself is a platform. The objective of this paper is to position OW2 as a *business ecosystem platform*.

In the first part, we look at the concept of the platform as it is usually applied to products and services, through examples from the IT and other industries and, in the second part, we explain why the platform concept also applies to OW2 and the tactics used to implement our platform strategy. Naturally, this paper is biaised toward technology and, more specifically, software.

[1] Initially published in Quaderni di Management (Italy) n° 33, May-June 2008 under the title "Reflexions on the OW2 Consortium Business Ecosystems Strategy".

-A- Characteristics of Platforms in Business Environments

1. Defining a platform

First of all, how do we recognize a platform? Why are some products or services called platforms? Is this purely marketing or can specific characteristics be identified which point to the platform concept? Different examples help us to identify the four key characteristics which, in our view, define a platform in a business environment.

Political platform

In politics, a platform often defines an ideal of citizenship and government. For instance, in the late nineteenth century, the reconstruction of America after the Civil War saw the opposition of the Democrat and Republican platforms and eventually the development of a national identity based upon a mainstream vision shared by all Americans: "(...) the Liberal Republican platform defined a new alliance in American politics for the rest of the century and began the reconciliation of North and South around the idea of individualism." [2]

First characteristic, a platform provides a *value that can be shared by different independent stakeholders.*

Product platform

Platforms have been well known for years in the automotive industry as a way to rationalize manufacturing: "For cars, the platform is primarily its chassis (...) and an associated family of engines and transmissions. Auto makers then create variants by putting different bodies on top of the same platform allowing multiple cart models to use it."[3] Product platforms are not exclusive to the manufacturing or high-tech sectors: credit cards companies, hotel chains and theme parks for example, also have core product platforms from which they develop a range of different services targeted at different market segments.

[2] Heather Cox Richardson, West from Appomattox: The Reconstruction of America After the Civil War Yale University Press, 2007, Page 147.

[3] Daniel F. Spulber, Global Competitive Strategy, Cambridge University Press, 2007, page 92.

Second characteristic, a platform groups together the *key elements common to a family of different products or services*[4].

Technology platform

In the world of technology a platform is usually at the crossroads between technologies, products and even stakeholders. Take, for example, the i-Mode technology for multimedia services in Japan: "The i-Mode technology platform consisted of two standards for creating content and for transmitting data wirelessly"[5] or in a more obscure field we find: "We studied a USN middleware platform based on multi agent (...) with standardized interfaces between wireless infrastructure and application services using multi-agents." [6]

Third characteristic, a platform *facilitates the cross-integration of third party offerings*.

Software platform

Platform is a popular concept in the software industry, consider the following excerpt from tyhe Eclipse Foundation: "(...) the project's broader goal is to deliver a general-purpose application and tool-integration platform. In order to fulfill this goal, it must be capable of integrating new functionality from different independent software vendors (ISVs) while preserving the appearance of a single cohesive environment"[7].

Fourth characteristic, a platform *provides and retains its own rationale and architecture when integrating complementary offerings.*

[4] Michael E. McGrain, Product Strategy for High Technology Comoanies: Accelerating your Business to Web Speed, McGraw-Hill; 2nd ed., 2000, page 53.

[5] Annabelle Gawer, Michael A. Cusumano, Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation, Harvard Business Press, 2002, page 215.

[6] Ngoc Thanh Nguyen, Geun Sik Jo, Robert J. Howlett, Lakhmi C. Jain, Agent and Multi-Agent Systems: Technologies and Applications: Second KES International Symposium, KES-AMSTA 2008, Incheon, Korea, March 26-28, 2008, Proceedings, Springer, 2008, page 692.

[7] Jim D'Anjou, Sherry Shavor, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy, The Java Developer's Guide to Eclipse, Second Edition, Addison-Wesley, 2004, page 219.

2. Three key platform mechanisms

Because of its characteristics, a platform can have an impact on its environment through three key mechanisms: the rationalization and the reduction of R&D and manufacturing costs, its extension through complemenary products and services and standards, and the support for firms inter- dependencies and business ecosystems.

Rationalization and the reduction of R&D and manufacturing costs

As seen in many industries, a platform can be the pivot of a product portfolio management strategy and help minimize the costs of developing product lines. For example, a platform strategy allowed, the French car manufacturer PSA to produce 85% of its cars on just three platforms.[8]

In a platform-driven R&D organization, where different products share the same structure, technology and production process, one can define *platform efficiency* as the "ratio between the average R&D cost (or develoment time) for derivative product over the cost (or time) spent for the platform."[9] This ratio depends upon the degree of commonality within a product family: the lower the ratio the more efficient the platform is at supporting the development of derivative products sharing the same components.

In business terms, the benefit of an efficient product platform strategy is the ability to design a family of products, or services, better suited to the specific needs of differents market segments setting the stage for a potential increase in revenue.

One platform, however, cannot be the universal answer to a product strategy; a given platform comes with its own limitations, it determines precisely the market positioning of an entire product line and its boundaries cannot be stretched too far. It has been noted that: "For instances, excessive component sharing accross brands in the automotive sector has often been criticized by consumer and the press as in the case of Ford components being used in

[8] Brüggemann/ Nyström/ Kiefer/ Gence, Competence Analysis: An Approach to a Firm gs Competence Domain: An Approach to a Firm gs Competence Domain, GRIN Verlag, 2007, page 12.

[9] John Clarkson, Claudia Eckert, Design Process Improvement: A Review of Current Practice, Springer, 2005, page 416.

Jaguar cars, or Volkwagen's use of the same platform for widely different models."[10]

Standards and extension through complemenary products and services

A platform's value is enhanced, on one hand, by derivative products provided by the platform developer and, on the other, by complementary product extensions or services offered by third party vendors. A platform offers opportunities for outsiders to develop profitable activities without having to develop the whole platform. From this point of view, a platform can benefit independent firms, outsiders or niche players.

Standards are important in this context because they facilitate (or reduce the cost of) the relationships between the platform provider and third party vendors and users. While product platform strategies developed in the automotive and aerospace industries were often based on proprietary standards (i.e. technical specifications owned and controlled by a single vendor and accessible only at a cost by third party vendors), platforms in the IT industry tend to be more reliant upon open standards which are, by definition, easily accessible to third party organizations.

Standards are important because they provide shared references and some long-term stability. Whether imposed to others through the dominant market market position of a powerful vendor or the outcome of a collaborative effort by engineers and standards organizations, standards help reduce the variability of base components and architectures. They tend to limit complexity and uncertainty and to lower barriers to information access, technical compliance, operational implementation, maintenance, service development, etc. They facilitate integration and, over time, help develop ecosystems of plug-and-play offering providers and achieve industry-wide linkage between stakeholders.

Firms inter-dependencies and business ecosystems

In today's fast changing markets where competitive advantage derives from rapid innovation and narrow market segmentation, market leadership is

[10] John Clarkson, Claudia Eckert, Design Process Improvement: A Review of Current Practice, Springer, 2005, page 416.

often achieved through a combination of a platform-based product development strategy[11] and an efficient leverage of complementary offerings. The management litterature abounds in examples of how successful platform strategies define industry structures and yield market leadership.

In high-tech sectors, it is well known that end-results are so complex that no one company can provide all the constituents required to fully address customer needs. In network industries such as such as telecommunications, software and hardware networks, utilities, banking services, etc. companies must necessarily take into consideration the providers of complementary products and services. "A firm must innovate internally to succeed - yet its success may equally depend on corresponding innovations by external firms."[12] In this context, the most successful companies are those who manage to position themselves as providers of the core technology foundation, i.e; the platform, shared by, and necessary to other products, services and solutions. A platform provider, its complementors and the network of reciprocal dependencies between them form an industry structure that we call a business ecosystem. "Platform serves as an embodiment of the functionality that forms the foundation of the ecosystem, packaged and presented to members of the ecosystem through a set of interfaces. Ecosystem members then (...) think of them as the starting point of their own value creation."[13] Platform providers become the keystones of their business ecosystem to the extent that "platforms provide them with a critical opportunity to shape and control their ecosystems."[14]

3. Multi-sided Platforms

Those platforms which serve as a hub for the relationbship of business partners of different natures are called multi-sided platforms. They are particularly efficient at structuring industry segments. Understanding how multi-sided platforms work will help us understand OW2.

[11] Timothy W. Simpson, Zahed Siddique, Jianxin Jiao, Roger Jianxin Jiao, Product Platform and Product Family Design: Methods and Applications, Birkhäuser, 2006.
[12] Annabelle Gawer, Michael A. Cusumano, Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation, Harvard Business Press, 2002.
[13] Marco Iansiti, Roy Levien, The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability, Harvard Business Press, 2004, pages 148-149.
[14] Ibid. page 158.

The multi-sided platform defined

A platform by itself is usually not enough to provide a solution; it requires complementary products and sevices. When they are provided by the platform vendor itself, we have a single-sided platform. When they are provided by independant third party vendors we have multi-sided platforms. To attract independant complementors, a platform must have a certain market reach or market recognition, but it can only acheive this if the collective value offered by complementors make it attractive for customers: the more customers the more complementors and vice-versa; this chicken and egg situation is fundamental of two-sided platforms. "Many if not most markets with network externalities are characterized by the presence of two distinct sides whose ultimate benefit stems from interacting through a common platform."[15]

There are many examples of two-sided platforms including shopping malls and software platforms: "The mall is available to stores and shoppers. Once there, the merchants and consumers interact directly on the platform. (...) Likewise, the software platform is available to developers and users. (...) Both user and developer rely on the services provided by the platform".[16]

Conditions for multi-sided platforms

Multi-sided platforms tend to be found in sectors or markets with three main characteristics.[17]

First of all, there should two or more distinct groups of stakeholders with a need for connection. This includes vendors seeking buyers such as, for example, merchants and credit card holders, but also vendors of complementary products or services willing to cooperate such as software and hardware vendors supporting a specific Linux distribution.

Second, there should be some industry-wide inefficiencies (or high transaction costs) in the connection process against which the platform represents a better solution, including through the provision of standard

[15] Jean-Charles Rochet, Jean Tirole, Platform Competition in Two-Sided Markets, Journal of the European Economic Association, vol. 1, n°4, juin 2003, pages 990-1029.
[16] David S. Evans, Andrei Hagiu, Richard Schmalensee, Invisible Engines: How Software Platforms Drive Innovation and Transform Industries, MIT Press, 2006, page 53.
[17] Ibid. page 55.

interfaces or processes, for example, eBay was created to help connect vendors and buyers of second-hand goods.

Third, there should be network effects or advantages in numbers as in the video game sector where "users like platforms with more games, and developers like platforms with more users"[18] which, in turn, would favour cooperation between complementors.

Skewed pricing in multi-sided platforms

In most cases, the cost of using a multi-sided platform is not evenly shared between the different groups of users: "In a N-sided market, price should be set so that the right communities are attracted to the market in the right combination and balance. (...) The essence of the argument is that in an N-sided market, the value obtained by each type of customer depends on the presence of other types of customers."[19] But stakeholder groups are not symmetrical and, technically, the way to obtain the right balance is to reflect these assymmetries in the platform pricing strategy: one group will end up paying more than the others. The group who will pay the most generally has one or more of the following characteristics: a) is most in need of the presence of the other group(s), b) has the lowest price elasticity and c) is easier to invoice by platform operators.

That is, simply put, the reason why most multi-sided platforms operate with a pricing structure that is strongly skewed toward one group of stakeholders. For instance, advertisers not readers bear most of the cost of newspapers, merchants not shoppers support the cost of shopping malls, etc.

Multihoming

There is generally no exclusive relationship between a user and a platform. The situation in which a user or a complementor joins or supports more than one platform for a comparative need or function is described bythe term *multihoming* which is borrowed from the vocabulary of computer networking technology. "For example, consumers may carry, and merchants may accept,

[18] Ibid. page 138.

[19] Marco Iansiti, Roy Levien, The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability, Harvard Business Press, 2004, pages 199.

more than one credit card for payment. Computer users may install a Windows or a Linux operating system on their PCs, or both. Software developers may write applications for Windows, Linux, or both."[20]

Multihoming behaviours depend on a number of factors (switching costs, platform differentiation, access cost, etc.) and contribute to shaping both competition and cooperation between platforms.

Feature accretion

As already noted, platforms by themselves do not provide any benefit to end-users. For a platform to be of use, it needs to get all stakeholders on board and to attract them it needs to offer them a credible value proposal embodied into valuable services. These can be shared facilities as in the case of the shopping mall (parking, restrooms, security, lighting, cleaning, etc.) or functionalities as with software platforms (graphic features, file management, resilience, ease of administration, etc.). Feature accretion is typical of software platforms: "And, as with all code-based products, they compete by adding features — and thus grow larger — over time"[21].

Sometimes platform grow to the point where they integrate features already offered by complementors thus jeopardizing their market positioning: "As platforms grow in the range of functionality they support, and as once-new functionality becomes increasingly stable and tightly integrated into the platform, firms that staked out terrain at the frontier of the platform will be absorbed as the frontier shifts outward. This is the fundamental reason why, for example, so many software middleware firms have failed to scale as independent entities."[22]

-B- OW2 as a Multi-sided Platform

OW2 is a consortium dedicated to developing a code base of open source middleware. As a non- profit organization, OW2 drives a community of developers and companies which share the same interests.

[20] Justus Haucap, Ralf Dewenter, Access Pricing: Theory and Practice, Emerald Group Publishing, 2006, page 230

[21] David S. Evans, Andrei Hagiu, Richard Schmalensee, Invisible Engines: How Software Platforms Drive Innovation and Transform Industries, MIT Press, 2006, page 216.

[22] Marco Iansiti, Roy Levien, The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability, Harvard Business Press, 2004, pages 154.

1. Market environment

Middleware defined

Middleware is commonly defined as the software layer that runs above the operating system and the network and under the application. Middleware offers a number of services through application programming interfaces (APIs) wich programmers use to develop their applications. We should add here that *application* means *business application;* to that extent a generic application such as a browser or a portal which offers APIs can be considered to be middleware. Essentially generic, middleware does not embody a business process nor the key business functions that make one company more competitive than another.

The middleware market opportunity

Our understanding is that market trends call for open source infrastructure software. Middleware becomes critical as corporations and public administrations open up applications to remote employees, partners, suppliers, customers and citizens, as distributed computing infrastructure interconnect a growing number of organizations, and as cloud computing emerges as a new key trend in computing.

Modern computing systems are increasingly complex and middleware, an essential part of the foundation of large information systems, has become a strategic infrastructure component of our information society.

Because the value of infrastructure software increases with usage (also known as "network effects"), a large open single source is more efficient than many smaller proprietary sources. Moreover, as middleware becomes generic, product differentiation in this market tends to become less and less strategic. The rise of open source software is consistent with such trends, as is the emergence of a concentrated supply of open source middleware software. The OW2 Consortium was founded to leverage these trends.

Leaders, outsiders and long tail

As it happens in all markets on their way to commoditization[23], the middleware industry is increasingly concentrated: standardization and company

[23] A future paper will discuss commoditization and its relevance to the OW2 community.

mergers and acquisitions are reducing the number of middleware vendors to a few, usually North American, companies. At the same time, however, continuous innovation in middleware on one hand and open source, as both a collective development organization and a market entry strategy, on the other support the emergence of new vendors.

Middleware covers a number of functionalities which are often incorporated into established software platforms and, as noted above, this can make difficult the growth of independent middleware vendors. While market leaders clearly control their own market environments (partners, complementors, even customers) outsiders and niche players struggle to control their own development paths. We think there is an opportunity to serve outsider vendors with modest market shares and fledgeling new entrants. We think that OW2 can play the role of the *aggregator*, making them "available and easy to find, typically in a single place."[24] There is a possibility that independent vendors coordinated around OW2 and abiding by mainstrean open standards can, collectivelly, gain enough market power so as to balance market concentration forces and the influence of proprietary market leaders.

2. Community structure

What is the structure of the ecosystem that surrounds OW2 and how do we segment the different groups of stakeholders we want to serve or attract? We have identified nine groups.

Industry leaders

Leading industry players include large companies who have, or who are planning to develop an open source strategy. Usually they are not pure play open source companies. They want to implment an oen source strategy on oine part of their activity and would join OW2 if they recognize that the consortium can help them. They intend to leverage their OW2 membership to grow their own market and business ecosystem. Expected benefits include: greater visibility and goodwill in open source developer communities, a platform for ecosystem development, increased market power and market share, and the creaion of de facto standards.

[24] As defined in Chris Anderson, The Long Tail: Why the Future of Business is Selling Less of More, Hyperion Books, 2008, page 89

Within OW2 this group includes companies such as Bull, France Telecom, Red Hat and Thales.

Software vendors, ISVs, start-ups

This group includes the many smaller companies which typically represent the most pro-active part of the open source market. They are innovative companies entirely dedicated to an open source business model. They join OW2 to become part of a business ecosystem which they expect to provide them with business opportunities and relevant feedback to accelerate the development of their technologies. These vendors expect networking benefits and they leverage the consortium to share the efforts required to build market visibility.

This group includes companies such as EBM Websourcing, Exo Platform and Xpernet.

Systems integrators

Most systems integrators are agnostic as it is not in their interest to specialize too narrowly in one kind of technology. Systems integrators join OW2 mainly because: they want to open source software modules that they often re-use in customer projects in order to share their development and maintenance; they have identified open source as a valuable market positioning tactic; they want to belong to a technology-oriented business ecosystem which can help them grow their revenues. In short, joining OW2 helps them boots the efficiency of their open source strategy.

This group includes companies such as CVIC-SE, Edifixio, Engineering Engneria Informatica, European Dynamics, Serli, SERPRO and Sogeti.

IT consulting firms

While systems integrators have the ability to take full responsibility for large projects, consulting firms concentrate on the initial phases of IT project life cycles: analysis, specification, technical expertise, architecture recommendations, etc. Typically they are small firms, even micro enterprises run by high-level consultants who leverage their community relationships to develop their business and maintain state-of-the-art expertise.

This groups include companies such as Altic, Artic Park, Experlog, Konsultex and Neociclo.

End-users

End-users are defined in opposition to other groups. They include companies which cannot be included in other groups, they do not sell software, SaaS (software as a service) nor IT services, and they do not develop software to be embedded into products. End-users, however, have development teams for their own projects which can contribute to the OW2 code base. They would join OW2 because the freedom derived from open source software has value for them, because they seek to share experience and best practices with other end-users and to have easy and privileged access to technical specialists in the community.

This group includes companies such as France Ministry of Interior, Hospitals, Banks, Utilities, etc.

Research organizations

These are private or publicly funded organizations with leading IT research teams developing a significant amount of code and are keen users of and contributors to open source software. These organizations are interested in developing relationships with the IT industry they are seeking real- life experiences and test opportunities; resources to fund research projects while permanently looking to enhance their reputations and visibiliy.

This group includes organizations such as CNRS, Fraunhofer, INRIA, ISCAS and GMRC.

Universities and IT R&D laboratories

This group is formed by universities and their research labs involved in software development. Students and post-graduates find in the OW2 code base a valuable environment for their studies. OW2 also provides them with an access to the open source world and to career opportunities. Most often, joining OW2 is the decision of a research laboratory rather than the whole university.

This group includes organizations such as Beihang University, Charles University, NUDT, Pekin University and University of Fortaleza.

Individuals

Although OW2 is an organization of organizations, it welcomes individual members. Many are freelancers and technology enthusiasts who have joined by curiosity or to improve their skills, to enhance there professional profile and some to find job opportunities, and some have registered on behalf of their employers in order to evaluate the opportunity of joining in the future.

Contributors

Contributors come from a variety of backgrounds. There are essentially three categories: the first includes direct projects team members, the second IT professionals using OW2 software in their own work (ISVs, systems integrators or end-users projects) and the third students working on projects within the framework of their studies.

3. Value proposal

All the above groups can consider joining OW2 because they have an interest in open source, middleware and the portfolio of software they find at OW2. But OW2 is much more than just a repository of open source code. It is an active organization whose mission, as written in its bylaws, is to "to develop open source middleware and to foster a vibrant community and business ecosystem"[25].

The OW2 Consortium operates for the benefit of its community. It is an organization designed to facilitate inter-relationships, first, between the community members themselves and, second, between the community and the market. To some extent, OW2 incorporates, on behalf of its members, functions, such as software distribution, license management, communication and evangelization, which are traditionally integrated into the value chain of independent companies. Or, at least, OW2 tends to share some of these functions with its members.

[25] OW2 bylaws: http://www.ow2.org/xwiki/bin/view/MembershipJoining/LegalResources.

OW2 provides three types of services to its community. The consortium is first a technical platform delivering collaborative services to project teams, second, it is a catalyst for social and business interaction, and third the consortium provides communication and branding services for developing projects' visibility and market awareness. The following paragraphs provide an overview of the services provided by OW2 to its members.

Technical services

The OW2 platform offers a range of technical services to its members. Architecting, implementing and running a collaborative development infrastructure are the fundamental services offered to the community.

These services are currently supported by several main applications. The first is a forge, the application which technically supports the projects through a number of tools to manage code contributions, versions, debugging, licenses, contributors, donwloads, etc. The second application is a mailing list system which helps create the lists used by the community. The third is a wiki system to support the organization's web site as well as projects' web pages.

Just as software platforms evolve by adding new features[26], we plan to keep adding new services and, over time, we have committed to adding features required by the different groups of stakeholders. The next wave of additions will include more tools for developers, a webinar solution to help members provide online presentations, a new forum to support more agile community interaction, a software appliance integration utility to enable our members to quickly integrate packages and a licence management system to improve the legal tracebility of the software on the OW2 code base. Not all the new services will be operated directly by OW2, some will be outsourced to third party providers and not all of them are clear open source followers. For each new service, the issue arises of whether OW2 should rely exclusively on open source software or whether some facilities can be provided by non open-source software and services.

[26] David S. Evans, Andrei Hagiu, Richard Schmalensee, Invisible Engines: How Software Platforms Drive Innovation and Transform Industries, MIT Press, 2006, page 305

Governance services

Although the open source development model has been compared to a "bazaar" as opposed to the well architectured, but rigid, cathedral[27], open source organizations such as the Linux, Apache and Eclipse foundations, and the OW2 Consortium aim to bringing a bit of the cathedral to the bazaar. The governance system implemented at OW2 relies on five guiding principles: Openness, Fairness, Trust, Transparenc and Independence. It provides a framework, rules and organizational entities designed to address three main risks perceived by the market. Technology risk: Is the code good enough? Does it do what it is meant to do? Is it safe?, etc. Legal risk: Do I infringe someone's rights when using or modifiying this code? What are my rights? Market risk: Is my investment protected? Will this code be supported in the long run? It takes a mature open source organization to address these risks, they are not covered by simple open source code repositories.

Let's take one of our governance entities, the Technology Council, as an example. It discusses the technical vision and controls the consistency of the code base, it recommends new projects, evaluates projects' progress and categorizes them into three evolution stages – Incubator, Mature and Archive – and it organizes discussions in the event of conflict between projects, etc. Some may think it is a bureaucratic organization but its role is to create consensus among members. An example of consensus is that a project can only be considered mature if is is documented, if professional support is available to help end-users implement the technology and if they have been tried and tested by community members. Projects in the OW2 code base can generally be supported by more than one members and are not entirely dependent on just one company as already proven in the past.

Marketing services

The role of the OW2 Consortium is also to build the community identity and brand and to help build the visibility of projects. In other words, OW2 drives a collective marketing effort to develop the visibility and market attractiveness of the community. While being developed by and for the

[27] Eric S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly, 1999, page 30.

Consortium, marketing and communication services benefit all members. This is achieved through a number of marketing and communication initiatives which, while developed for the group, help members improve their own market positioning. Indeed, the more the Consortium communicates, the more it gains respect and the greater the market recognition of its members.

With the help of marketing specialists from the community, the Consortium's Management Office (i.e; the day-to-day management team of the organization) supports the community's communication essentially in three ways: creating collateral, organizing the community's presence at professional events and driving outbound communication.

Collateral include presentations, a Web site, flyers, project datasheets, case studies, white papers, factsheets, logos and other tools such as flyers, T-shirts, note pads, pens, stickers, etc. Promoting projects at conferences and events involves, for instance, circulating call for papers, coordinating the participation of members at major trade shows under a single OW2 banner, organizing specific sessions to present OW2 projects in public conferences or setting-up public community meetings such as the OW2 Annual Conference, etc. Outbound communication efforts include the writing of press releases, presenting OW2 at conferences, producing a monthly newsletter or a blog, advertising OW2 in community-related web sites or paying to drive traffic to the OW2 Web site and briefing industry analysts and journalists, etc.

4. Pricing strategy

OW2 is a non-profit association and, to protect its independence, it relies exclusively on membership fees and seeks neither governement subsidies nor participation in publicly funded projects. As a result, the organization is entirely dependent on its members renewing their commitments. Should OW2 cease to provide adequate value for the money paid by its members, they would leave and the Consortium disappear. Genuine commitment to provide positive return on investment (ROI) to members and the need to attract all groups in the community have lead to the implementation of a carefully segmented and skewed fee structure. The OW2 pricing strategy has four characteristics.

First of all, membership fees cover *access fees*; they are not usage fee. In a way, members pay for the right to use the Consortium's technical infrastructure (whether they use a lot of bandwidth or not does not affect their fees), to participate in its governance system (i.e. to participate in the decision- making

process) and to leverage the brand (i.e. to increase their own market power). They also pay to guarantee the sustainability of the Consortium. This is not to say that there are no variable costs for members: they can be significant and consist of all the costs, mainly staff and travel, incurred by participating in the consortium's activities. These costs represent the real *usage fees* members should consider. But it is understood that members will only support these costs because when they participate in a Consortium's activity, they do it for their own benefit.

Second, the Consortium derives most of its resources from one group of members, the Strategic Members[28] group. This group includes industry leaders and leading research organizations which have decided to leverage OW2 for the development of their own open source strategies on the promise that it would cost less to do so than going through the learning curve of building a community and an open source organization from scratch on their own. Compared to other groups of members, the Strategic Members have a long-term commitment to the Consortium, they stand out to provide significant resources to support the Consortium's objectives and wish to play an active role in setting the directions of the Consortium's in both the code development activities and facilitating the use and acceptance of the Consortium's technology. This is the reason why Strategic Members commit to remain members for a minimum of three consecutive years.

The third characteristic is that, for Corporate Members, in between those who pay the most and those who pay nothing, the pricing scheme is designed to closely match the resources available to members. There are three company segments: large organizations and small and medium size organization as defined by the European Commission, micro organisation for companies under ten people, and two academia segments: universities and research laboratories, the main difference being that the former may have thousands of students while the latter only a few dozen. Interestingly, these segments were not defined ex-ante but in response to the requests made by potential members. One category of members, Individual Members, can actually join for free; that is because the price elasticity of this group is very high and a strict ROI analysis

[28] The OW2 membership is divided into three categories: Strategic Members, Corporate Members and Individual Members. See: http://www.ow2.org/xwiki/bin/view/MembershipJoining/MembershipCategories.

would lead to a fee so low that the financial gain for the Consortium would probably be offset by the cost of collecting the fees. There is an interesting debate whether end-users could be members at no charge for specific reasons;

Fourth, and last but not least, in order to comply with one of our five guiding principles, Fairness, in our pricing strategy, we decided to apply the Purchasing Power Parity ratio[29] as defined by the World Bank so that the financial effort of membership would be the same in rich as in developing countries. This is another way of implementing a skewed pricing structure to adjust to members expectations. Corporate Members who compete at a local level can be expected to support a membership fee compatible with the cost of living in their countries, Strategic Members however compete at a global level and it can be debated whether it is relevent to calculate their fees using the Purchasing Power Parity ratio.

5. Platform governance system

Confronted with the rise of open source, established commercial vendors have often retaliated by deploying FUD (Fear, Uncertainties and Doubt) communications strategies. Targeted at fledgling offerings by communities and open source vendors, these strategies have been devastating. The cultural image attached to open source is still by and large a liability to the point that, if commercial software is simply evaluated on the basis of standard buying criteria, an open source proposal needs to provide additional guarantees. The open source environment is still perceived as a jungle by a number of decision makers and the conventional information system manager often feels uncomfortable with the open source model which is by and large still perceived somewhere between rebel and immature[30].

To counter this perception we have implemented a simple yet effective governance system materialized by governing bodies with clear roles. At OW2, members work collaboratively at developing the code base and their business and community relationships within the framework of three kinds of activities: *Projects*, *Initiatives* and *Local Chapters*. Decisions are made at three levels: the Board of Directors discuss the strategic decisions, the Management Office (MO) makes the decisions relevant to running the organization and each

[29] http://www.ow2.org/view/MembershipJoining/ppp.
[30] Source: private interviews.

Activities Management Team makes the day-to-day decisions to its activity. The Board and MO can rely on three Councils available to provide expert guidance and recommendations. The Technology Council for issues related to the technology platform and the code base, the Ecosystem Council for marketing and communication and the Operations Council for administrative, legal and financial affairs. At this point however, only the Technology Council is running properly.

It is not a one-sided leadership as in most software industry technology platforms. The OW2 governance system is democratic: all decisions are transparent and can be challenged and discussed.

The Intellectual Property Right (IPR) policy is a key part of the governance system. OW2 IPR policy is threefold. First of all, at OW2 we decided not to contribute to open source licence proliferation (we did not create the OW2 public licence) but to rely on existing open source licences. Second, we have implemented a mechanism called revocable non-assertion by which a company can bring patented code to our code base under a open source licence and still be protected. Third, because we do not want to restrict our members ability to develop a profitable business derived from its open source contributions, we accept dual licencing.

Governance drives the evolution of the code base and of the services provided by the OW2 platform. OW2 is a community driven by a set of rules and governance systems which aims at minimizing the randomness of relational power networks and politics often found in grassroot communities. From this point of view, OW2 is very much comparable to other open source organizations such as the Apache, Eclipse and Linux foundations even if some of its options, its IPR policy in particular, are different. Currently, it could be said that, of all these organizations, Eclipse is the benchmark against which OW2 evaluate the efficiency of its governance system.

-C- Specific Challenges

1. Multi-homing and self-homing

ISVs and Systems Integrators support OW2 typically because the consortium provides them specific services. In paying their fees, they form a community sharing the same commitment to building and sharing the tangible and intangible assets of the consortium. The scope of services provided by

OW2 does not claim to address the entire spectrum of their activities. It therefore happens that some firms either look for additionel services from different organizations, buy them from private vendors or develop them themselves. Example of such services include: recruitment, market studies and marketing services, product management, etc.

It happens in many service business that the service provider risks to finds itself competing with its own customers; this rule applies to OW2. All the situation exist from very small companies depending extensively on OW2 to firms with such critical mass that they border on finding more valuable to creat their own community environment. When companies develop their own community assets we can talk of *self-homing*. Self homing may have the well known advantages of hierarchies[31], the principal one probably being the marketing ability to better track and manage user behaviors in order to increase conversion rates to services for a fee, but it must be very well managed to avoid frictions in the world of open source communities. Any member tempted by self- homing on the basis of recent success must clearly evaluate whether this decision will bring a long- term competitive advantage.

Besides a possible combination with self homing, there seems to be little incentive for multi- homing. We see no particular strategic or marketing advantage in supporting two, or more, platforms because this reduces the network effects of concentrating its commitment on one platform. Multi-homing can be justified only by a clear complementarity – rather than differentiation – between community environments. For this reason some OW2 members are also Eclipse members, for instance, but not for the same projects.

One last comment. There seems to be significant switching costs between community platforms and moving to another community environment is not an easy decision to take. For a member to leave a platform or to switch to another one is generally the result of a painful or radical decision driven by, for example, a member's internal change of leadership and strategic realignment, a management failure to execute the strategy which initially lead to joining the platform, a member's management complete disagreement with orientations given to the platform.

[31] Oliver E. Williamson, Markets and Hierarchies: A Study in the Economics of Internal Organization, Free Press, 1983, page 257.

2. Opportunism

OW2 is founded on a simple contract between members. The platform was established as a non- profit association with the following purpose: "to develop industry grade open source middleware, to nurture the associated code base, to foster cooperation among its Members, and to help foster a vibrant eco-system for the exploitation of its middleware code base." [32] It is also said that the activities "shall not be conducted for the financial profit of its Members but for their common benefit."[33] A simple contract, yet its execution seems to be hampered by a number of behavioral issues.

The OW2 platform is driven by a limited group of stakeholders who are not immune to opportunistic behaviors. Such behaviors, well described by economists[34], typically occur when agents believe they can bend some shared rules in pursuit of their own interests and escape detection and retaliation. "There is a considerable gap between the desire to engage in co-ordinated interaction and the ability to do so successfully. It can be difficult to reach mutually acceptable terms of co-operation, and to ensure that firms do not deviate from them"[35].

These issues derive from the fact that OW2 is endeavouring to organize the interdependence of agents characterized by a mix of converging and diverging interests. Members have entered into the OW2 contract with different perspectives: for instance, some are commercial companies whereas others are academic institutions, some are large, diversified organizations whereas others are small, single product companies, some are financed, others struggling, etc. Moreover, some have joined as strategic members and others as corporate members. It is therefore not surprising that members have different attitudes towards the consortium.

This difference is manifest in the way members comply with their commitments, specifically strategic members who have the highest level of commitment. In return for governance privileges, strategic members commit to providing both financial and in-kind contributions – personnel resources to

[32] OW2 Bylaws, Section I.4 Purpose.
[33] ibid.
[34] George J. Stigler, "A Theory of Oligopoly," Journal of Political Economy, 22 (1964), pages 44-61
[35] Directorate for Financial, Fiscal and Enterprise Affairs, Committee on Competition, Law and Policy, Oligopoly, DAFFE/CLP(99)25 (1999), page 7.

help run the consortium to the extend of a full-time equivalent – for a period of three years. Information on financial contribution is clearly accessible: annual dues have been settled or not. In contrast, in-kind contributions are more difficult to measure. A questionnaire distributed to strategic members on their in-kind contribution produced the most improbable – and creative! – answers. The overall impression is that the majority of strategic members are opportunists in the way thay do not provide their in-kind contribution in full. Some strategic members are also opportunists in the way they threaten to break their three-year agreements after having enjoyed their governance privileges for two years. As far as corporate members are concerned, some can be regarded as opportunists in the way they use OW2 exclusively – and, for some, abusively – as a download infrastructure without contributing back to the community or taking part in the activities of the consortium (this is our version of the "free rider" problem).

The OW2 platform governance is characterizzed by two weaknesses. First, it is difficult to really measure in-kind contribution or how members take advantage of the technical infrastructure. Second, despite legal binding by the membership contract and the bylaws, the potential cost of international litigation make such action almost impossible. Imperfect information on one hand, and the difficulty to enforce the commitment on the other, fuels an inherent propension to opportunism.

This is a well known situation where members have a quasi structural incentive to "cheat"[36] [37] in order to maximize the net gains from their participation in the consortium. Developing OW2 is an on-going exercise in reconciling diverging and converging interests and in developing such a compeling value proposal that the incentive related to converging interests is permanently greater than that of diverging interests.

3. Maturing endogenous and collective leadership

The OW2 platform is an open source community organisation which belongs to its members and, because of this, it must be driven by a collective governance and not by any measure of strong leadership or dictatorship no

[36] George J. Stigler ibid.

[37] Oliver E. Williamson, Markets and Hierarchies: A Study in the Economics of Internal Organization, Free Press, 1983, page 257.

matter how benevolent[38] it might be. In this respect, the OW2 platform is very much unlike a product or a technology platform controled by the R&D and marketing despartments of a firm, such as Intel's X86 platform or Microsoft's Windows platform for example. The evolution of the OW2 platform reflects the maturation of the collective wisdom of the consortium's members.

As noted above, the OW2 platform brings together different types of stakeholders and their own goals in interacting with the platform may differ. When they join OW2, members bring their own experience and understanding of the open source movement. Heterogeneity, diverging expectations and incongruent objectives are problems well known to organizations.

As a platform, OW2 can exist only in so far that it offers its stakeholders benefits which exceed the contributions it requires from them and, as a community-driven organization, OW2 can exist only in so far as it offers proper conditions of reciprocity and equity to its members. We can analyse OW2 as a bureaucracies[39] which role is to organize the interdependence of its members more efficiently and on a fairer basis than provided by the market. However, as we have seen, because of excessive ambiguity in the evaluation of the contribution to the OW2 platform and the utilization of its resources, the bureaucratic organization of OW2 is still quite inefficient at preventing opportunistic behaviors.

We expect that the world of open source's strong ethical values will help contain opportunistic behaviors within acceptable limits. And we assume they are more the result of a lack of maturity in the OW2 platform or, in other words, of uncertain intentions in an uncertain environment, rather than deliberate strategies. From this perspective, a key success factor of the OW2 platform will be its ability to grow a common expectation ("goal congruence") among its members in order to be able to operate with a certain level of uncertainty, between tolerance and opportunism, on participants contribution and usage ("ambiguity in performance evaluation")[40] In the meantime, OW2 will have to better explain its strategy and educate its community or should this fail improve its ability to strictly enforce its bylaws.

[38] Eric S. Raymond, The Catedral and the Bazaar, Musing on Linux and Open Source by an Accidental Revolutionary, O'Reilly, 1999, page 124.
[39] William Ouchi, "Markets, Bureaucraties and Clans", Administrative Science Quarterly, Vol. 25, No. 1. (March 1980), pages 129-140.
[40] Ouchi, ibid, page 135.

Conclusion

Established from the onset as an unconventional means to share code among developers, open source has evolved into a major structuring factor in the software industry. Since open source is on its way to becoming mainstream, it is expected by the market to give itself the necessary business, community and legal environment to make it a sustainable phenomenon. The OW2 Consortium was launched to leverage these trends. OW2 is both an open source community and a community-driven organization. OW2 provides a meeting point for stakeholders of differing natures who share an interest, either technical or business, for open source middleware.

In this paper, we have analysed how the OW2 Consortium is positioning itself as the platform at the center of a business ecosystem. We have shown how OW2 is implementing the core tactics of multi-sided platforms including skewed pricing schemes and feature extension.

Today's priority for the OW2 platform is to explain clearly its positioning and to offer a vision, an ambition shared by all participants so as to, first, develop a real community momentum among existing members and, second, to become both visible and attractive enough to potential members.

Now that the OW2 Consortium is establishing itself as a community-driven business ecosystem platform, its next challenge is to actually offer a technology platform as defined at the begining of this paper. Today, projects in the OW2 code base follow their own architecture and integrating them sometimes requires significant efforts. An ambitious goal for OW2's should now be to offer a real technology platform with projects which seamlessly integrate with each other. Combining a consistent organization with a consistent technology would grant the consortium and its members a long-lasting competitive advantage in the ever-changing software industry.

RekZit – vencendo os desafios da captura e gerenciamento de requisitos com uma ferramenta livre

Leonardo Thomas Torres Santos
Márcio Lima Albuquerque
Sandro de Carvalho Franco

A utilização de processos de desenvolvimento pelos grandes centros de tecnologia da informação é um fator imprescindível na difusão das melhores práticas da engenharia de software. Uma das formas mais significativas de se medir o sucesso de um projeto de sistema está no grau de atendimento às necessidades do usuário com a solução concebida. Na *engenharia de requisitos*, o envolvimento do gerente de projeto, desenvolvedores, revisores, clientes e todas as partes interessadas nas atividades de levantamento de informações, documentação e validação são de grande valor para a construção de um software com sucesso.

Estabelecendo um comparativo entre organizações com um nível de maturidade compatível, a utilização de ferramentas de apoio é muitas vezes o grande diferencial competitivo. Ferramentas adequadas integradas ao processo da organização trazem grandes ganhos na eliminação de tarefas redundantes, reduzindo o tempo necessário para a elaboração da solução. Com isto, minimizam-se defeitos, diminuem-se riscos, aumenta-se a qualidade do produto final.

Este trabalho apresenta uma ferramenta de apoio ao desenvolvimento de software, focalizando a *engenharia de requisitos*. O intuito é de mostrar que é possível analisar e documentar os requisitos de software utilizando um sistema com características colaborativas, desenvolvido por analistas do SERPRO com as tecnologias mais recentes da *web 2.0*, utilizando plataformas

livres e com diversas vantagens sobre as ferramentas proprietárias atualmente empregadas. Serão comentados os ganhos observados, sobretudo na facilidade do uso e na possibilidade de adaptação total da ferramenta à realidade de uma empresa de grande porte, que possui um grande ativo de conhecimento, um processo em constante evolução e alinhada com as políticas de utilização e disseminação da tecnologia livre.

Cenário

Nas empresas públicas ligadas à tecnologia, as políticas de adoção do Software Livre[1] surgiram de necessidades naturais, seja na busca pelo conhecimento de ponta pelos técnicos, seja pela busca por soluções adaptáveis à realidade da empresa, passando também pela diminuição do custo por conta das licenças de software. Ressaltamos que, mesmo com todos estes fatores, o pretexto principal é a associação do Software Livre com a independência na escolha e o aumento da capacidade técnica necessário para utilização e adaptação das tecnologias livres. Outro ponto importante é a possibilidade de uma quebra de paradigma, pois seguindo a filosofia livre é possível desenvolver um software com qualidade contando com a ajuda de diversas comunidades distribuídas, de forma que todos se beneficiarão com o resultado final.

Inserindo o Software Livre na utilização de ferramentas de apoio ao desenvolvimento, encontramos muitos trabalhos interessantes que merecem destaque. A área de requisitos apresenta algumas iniciativas como a OSRMT[2], contudo, a maioria não atende a diversos quesitos, apresentando um grau de amadurecimento aquém do desejado (INCOSE). Aprofundando-se mais nesta questão, conclui-se que o levantamento e documentação de requisitos seguem um processo particular na comunidade livre, distanciando dos processos mais formais. No modelo clássico de desenvolvimento, a motivação para construção do software é originada de uma necessidade do negócio do cliente. No mundo

[1] Software desenvolvido segundo a filosofia GNU, que atende aos quatro graus de liberdade: liberdade de executar o programa para qualquer fim (liberdade n°0), liberdade compreender como o programa funciona internamente, alterando o seu código fonte de acordo com as necessidades (liberdade n°1), liberdade para copiar o programa (liberdade n°2) e liberdade de distribuir as benfeitorias realizadas para todos (liberdade n°3).

[2] OSRMT – Open Source Requirements Management Tool http://www.osrmt.com, uma ferramenta open source para gerenciamento de requisitos.

das tecnologias livres, a construção de novos softwares é impulsionada pelos ideais das comunidades, seja por *hobby*, gosto por desafios ou mesmo por diversão (AGUIAR, 2007). Estas particularidades acabam por refletir nas ferramentas adotadas, fazendo com que elas não atendam a um público que trabalhe com um modelo de desenvolvimento mais tradicional, firmado em termos contratuais, como é o caso da maioria das grandes empresas de desenvolvimento software.

Quando se volta para as soluções proprietárias, a suíte Rational[3] merece algum destaque pela sua popularidade. A IBM Rational apresenta um conjunto de softwares integrados que auxiliam nas diversas etapas do ciclo de vida do software. Dentre os aplicativos da Rational, destacamos o RequisitePro. Este software é restrito ao sistema operacional Windows, funciona integrado ao MS Office e auxilia no processo de *engenharia de requisitos de software*. Sendo facilmente encontrados em grandes empresas que seguem um processo baseado em RUP[4], voltadas para o modelo CMMi[5], os programas da IBM Rational atendem a diversas exigências do processo. No caso do SERPRO, que adota o PSDS – Processo Serpro de Desenvolvimento de Soluções - fortemente influenciado pelo RUP, a aquisição destas ferramentas tiveram grande valor como impulsionadoras para conquista da certificação CMM.

Porém, na medida em que a organização amadurece, as ferramentas de apoio ao processo devem acompanhar a uma série de mudanças necessárias. As ferramentas proprietárias, por sua vez, possuem o código fonte fechado, não permitindo modificações (no modelo proprietário, a forma mais comum de se obter atualizações é por compras de licenças). Já as ferramentas livres saem na vantagem, pois com o código fonte aberto, podem ser modificadas e adaptadas de acordo com as necessidades do usuário. Isto faz com que as ferramentas proprietárias sejam uma escolha ruim para as organizações, pois um dia não atenderão plenamente e não poderão ser adaptadas sem um custo adicional com aquisição licenças.

---

[3] IBM Rational http://www-01.ibm.com/software/rational/ empresa do ramo de Tecnologia que provê soluções de software

[4] Rational Unified Process – http://www-306.ibm.com/software/awdtools/rup/ é um processo completo que abrange todas etapas do desenvolvimento, proposto pela Rational e utilizado em inúmeras empresas de Tecnologia da Informação.

[5] Modelo de maturidade amplamente utilizado por empresas de Tecnologia da Informação. Indica o grau de desenvolvimento desta empresa na aplicação das melhores práticas para o desenvolvimento de software.

As aplicações proprietárias apresentam limitações ainda mais sérias na sua capacidade de integração. Isto acontece porque muitas vezes estes softwares não seguem padrões de interoperabilidade, adotando padrões fechados e próprios. Com isto, estas ferramentas só funcionam de forma integrada com outras do mesmo fabricante. Para manter a integração, as empresas são forçadas a adquirir a um alto custo (BNB, 2003)(TRTRJ, 2006) um pacote completo de software para a realização de um processo de desenvolvimento em todas as suas etapas.

Concluímos que, apesar da popularidade de algumas ferramentas proprietárias de apoio ao processo em grandes organizações, com o tempo é necessário se desfazer destas ferramentas. Isto se dá devido à obsolescência destes softwares. Hoje, empresas como o SERPRO, que num passado utilizaram-se de uma série de softwares proprietários, ainda passam por grandes dificuldades para se desfazer deles. O resultado disso é vivenciado como um caso de aprisionamento tecnológico por conta de um legado inconscientemente construído. Isto ocorre basicamente por dois motivos: (a) Os formatos de arquivo utilizados por estas ferramentas são incompatíveis com padrões abertos; (b) Não existe no mercado, ainda, softwares livres no ramo da *engenharia de requisitos* que atendam às necessidades específicas da empresa.

Motivação

A motivação para este trabalho veio deste problema real, no qual se tentou resolver de forma mais natural possível um desafio que atingira o dia-a-dia de inúmeras equipes de desenvolvimento e de gestores. Com problemas assim, em um dado momento, vem à tona uma proposta de solução. Compartilhando-se a esta idéia com grupos maiores e conquistando o apoio de pessoas que compartilham da mesma situação, chega-se a uma rede de colaboradores cada vez mais crescente (RAYMOND). A experiência vem mostrando que quando se assume a postura proativa diante de um problema que pode ser o de muitos, o resultado tende a ser o mais positivo possível, muitas vezes superando as expectativas iniciais.

Apresentaremos neste trabalho uma ferramenta construída com tecnologias livres, que foi implementada nas horas vagas e nos intervalos entre as demandas da empresa, bem como em momentos de tempo livre como um *hobby*. O objetivo inicial da ferramenta era servir como prova de

conceito[6], que buscava verificar se seria possível controlar os requisitos de software em uma ferramenta desenvolvida de acordo com as necessidades de uma grande empresa. Para isto, seria preciso seguir um processo estruturado com base no RUP. Esta ferramenta foi sendo aprimorada com base no processo, levando à incorporação de necessidades e funcionalidades extraídas do próprio PSDS.

Gradativamente, na medida em que foi se tornando funcional, a idéia foi sendo apresentada a grupos maiores, despertando grande interesse por entusiastas do modelo colaborativo de desenvolvimento. Por fim, o esforço desprendido nesta prova de conceito logrou êxito, saindo da condição de laboratório de testes e elevando-se para o patamar de realidade, terminando como uma solução plausível e conclusiva para um problema que fora levantado há muito tempo e sem uma resposta: que ferramenta livre adotar para apoiar a realização das atividades da engenharia de requisitos?

A dimensão da solução é considerada de grande abrangência e de grande impacto positivo nas áreas de desenvolvimento de sistemas e de negócio. Esta afirmação pode ser comprovada em dois aspectos mais importantes:

- Primeiro em termos de escala, a solução atinge a uma grande quantidade de usuários desenvolvedores (atualmente cerca de 3.100), justamente a maior parcela de usuários que pela falta de uma ferramenta no mundo livre, terminou por forçosamente utilizar a plataforma Windows para realização de tarefas exigidas no PSDS. Atinge também consideravelmente a parcela de usuários gestores, dos clientes finais e analistas de negócio, que poderão acompanhar a evolução dos requisitos de software de forma instantânea, interativa e transparente;
- Segundo, por estabelecer uma grande facilidade na forma de se criar e manter os requisitos documentados nos sistemas, favorecendo a integração entre ferramentas e avançando no aspecto do armazenamento de informações, que passam a ser recuperadas por meio de bases de dados e não mais por meio de documentos textuais eletrônicos. A capacidade de interatividade facilitada pela *web 2.0* (WIKIPEDIA) traz para o usuário final experiência de aplicativos mais responsivos, agregando aos *sites* características de aplicativos *desktop*, executados localmente.

[6] Prova de uma teoria, assegurando a veracidade de algo que fora proposto em modelos

Ainda não foram considerados os impactos positivos nos aspectos econômicos que esta proposta gera, uma vez que nem todos os dados encontram-se atualizados. Podemos apenas estimar uma grande economia pela dispensa de licenças de software e almejar possibilidades de comercialização da ferramenta.

Apresentados o cenário e a motivação do trabalho, faz-se necessária uma pequena explicação do conhecimento envolvido na construção desta ferramenta.

Engenharia de Requisitos

Requisitos de software são um conjunto de características ou capacidades que o sistema deve realizar para atender a uma necessidade do usuário. A *engenharia de requisitos* é área de estudo que agrupa um conjunto de conhecimentos e atividades voltados para a coleta e documentação de requisitos de software.

A *engenharia de requisitos* não se resume a um processo técnico. É uma área desafiadora, sobretudo por envolver também fatores humanos. O trabalho do analista de requisitos é considerado complexo, pois exige habilidades para compreender as reais necessidades do cliente. Sob inúmeras influências, ora mais presentes, ora mas sutis, que interferem na percepção do ambiente em que se encontra, o analista de requisitos precisa possuir um perfil investigativo e questionador para descobrir as informações servirão de fundamento para a construção do sistema.

Temos então os seguintes objetivos desta disciplina:

- Formar um consenso com os clientes e demais interessados sobre o que o sistema deve realizar;
- Fornecer para a equipe de desenvolvimento uma base que facilite a compreensão do negócio do sistema;
- Estabelecer as fronteiras do sistema;
- Gerar uma base que possibilite planejar os ciclos de implementação da solução técnica;
- Oferecer insumos para uma estimativa de custo e tempo para desenvolvimento do sistema;
- Definir uma interface do usuário para o sistema, focada nas necessidades do usuário final.

A seguir, serão apresentados os termos utilizados na *engenharia de requisitos*. Chegaremos então num conjunto de conceitos e atividades que juntos, definem a *engenharia de requisitos*.

Tipos de requisitos – sistema *FURPS+*

Segundo Grady, citado por (ELES, 2005) e (RUP), independentemente do software considerado, os requisitos podem ser classificados em uma das categorias do sistema FURPS+ (FURPS mais). Nesta classificação, FURPS é o acrônimo para os principais tipos de requisitos que são: **F***uncionality*, **U***sability*, **R***eliability*, **P***erformance*, **S***upportability*. Iremos detalhar cada uma destas categorias.

- Funcionalidade - *funcionality*: determinam um conjunto de funcionalidades que o sistema deve realizar. São os *requisitos funcionais*, que normalmente traduzem para o sistema parte do fluxo de negócio.

Para ilustrar o conceito de *requisitos funcionais*, podemos formular algumas sentenças como:

1. "O usuário deve poder incluir uma nova venda no cadastro, informando os detalhes da venda"
2. "O usuário deve ser capaz de realizar pesquisas no cadastro de vendas"

As categorias restantes, que formam o *URPS*, descrevem os *requisitos não funcionais* do sistema. Estes requisitos geralmente são resolvidos por questões arquiteturais do sistema.

- Usabilidade - *usability*: capacidades relacionadas com a facilidade no uso e a experiência do usuário. Tempo de aprendizado, ajuda *online*, fácil navegação pelas telas do sistema são exemplos de requisitos de usabilidade.
- Confiabilidade - *reliability*: são características que guiam o comportamento do sistema em caso de falhas, procuram garantir recuperação em caso de erros e a exatidão de resultados.

- Desempenho - *performance*: dão suporte a capacidades relacionadas com a rapidez do sistema e as otimizações no uso dos recursos.
- Suportabilidade - *suportability*: diz respeito a questões de adaptabilidade do sistema a outros ambientes, opções para configuração e manutenção facilitada.

O + do FURPS+ inclui outros requisitos geralmente ligados a restrições do sistema. São eles:

- Requisito de projeto: são todas as restrições que atuarão no projeto do sistema.
- Requisito de implementação: resumem como será o ambiente de implementação, constituído por uma linguagem de programação, meios de armazenamento, padrões de implementação, recursos e sistemas operacionais.
- Requisito de interface: especifica como será o contato do sistema com o mundo exterior.
- Requisito físico: diz respeito, sobretudo, a características físicas de hardware que devem ser consideradas no sistema.

Atividades da Engenharia de Requisitos

Conceitualmente, a engenharia de requisitos é exercida através de um conjunto de atividades principais que são:

- **Levantamento de requisitos**: tarefa inicial de descoberta dos requisitos.
- **Documentação e Análise:** registro da especificação do software em forma de textos, casos de uso, histórias do usuário ou modelos de processos. Refinamento, classificação dos requisitos, eliminação de problemas na clareza, conflitos e redundâncias.
- **Verificação e validação:** são processos exaustivos de testes, revisões e inspeções que garantem qualidade do produto (GROSSO, 2006). A verificação atua com intuito de conferir eliminar defeitos na especificação . Já a validação avalia se o produto especificado atende às necessidades do cliente (PSDS)(CMMI).

- **Gerenciamento de requisitos:** paralelamente a estas atividades, existe o processo de gerenciamento dos requisitos, que realiza o tratamento adequado para as mudanças que ocorrem nos requisitos ao longo do ciclo de vida do software.

Estas atividades estão descritas aqui em uma ordem seqüencial. Contudo, na prática elas estão entrelaçadas no processo (PSDS). Não há, portanto, uma fronteira explícita entre elas. Há sim, muita interação e interdependência entre todas as atividades. Abordaremos com mais detalhamentos as atividades aqui enumeradas.

1. Levantamento de requisitos

Nesta atividade, a equipe de desenvolvimento tem o primeiro contato com o negócio do sistema a ser desenvolvido. O foco está na descoberta de necessidades direcionam aquilo que o software deve realizar para satisfazer as expectativas do cliente. Com isto, é possível estabelecer uma visão comum sobre o escopo problema que precisa ser resolvido. O problema é então definido por escrito para que todos possam chegar ao consenso sobre o que está sendo discutido. Serão identificados todos os envolvidos, definindo o papel e responsabilidade de cada um. É preciso conhecer informações a respeito de como estes envolvidos interagirão com o sistema.

Serão obtidas as restrições do ambiente em que o problema se encontra, enumerando fatores externos de ordem tecnológica, política ou humana que interfiram no tratamento do problema. O fluxo de negócio do cliente serve como importante insumo para esta etapa. A saída são os requisitos iniciais que formam a visão do sistema, que servirão de principal referência para as demais etapas do ciclo de vida até a validação do produto final. A fronteira do sistema será definida, descrevendo o ponto de divisão entre o sistema e o mundo real. É nesta fronteira que o usuário interage com o sistema, realizando trocas de informações.

O trabalho de levantamento tem características colaborativas, com contribuição de pessoas que atuam em diferentes domínios. Desenvolvedores e analistas, responsáveis pelo negócio, cliente e usuário final trabalham juntos no sentido de conhecerem o fluxo de negócio, firmando uma visão comum do problema. As técnicas tradicionais para levantamento de requisitos são a realização de entrevistas. Independentemente da técnica adotada, a

abordagem do analista de requisitos pode intimidar os entrevistados, fazendo com que os fatores humanos sejam decisivos para o sucesso ou fracasso da técnica aplicada.

A elaboração de protótipos de sistemas enriquece este tipo de atividade (Nuseibeh, 2000), pois dá ao cliente uma visão de como a aplicação funcionará, dando uma prévia de como serão as telas e a navegação do usuário final. O protótipo pode ser visto como uma versão simplificada da interface do sistema, criado de forma a poder ser facilmente alterado e manipulado, até chegar a uma versão final, que possui interface que o usuário deseja.

Quando elaborado e apresentado ao cliente com os cuidados necessários, o protótipo é uma ferramenta poderosa na definição de sistemas. Isto corre porque tem um apelo maior do que definições textuais. Observa-se em muitos casos que as representações visuais e interativas prendem mais atenção do usuário, além de serem muito esclarecedoras. De forma empírica, percebe-se que em muitos é mais rápido e direto acessar uma interface visual do que interpretar textos longos e fluxogramas complexos.

Os requisitos levantados precisam ainda ser melhor documentados, pois se encontram em um estado mais "bruto". Esta lapidação ocorrerá gradativamente, quando a equipe realizar a documentação e análise destes requisitos (RUP)(PSDS).

2. Documentação e Análise

A forma tradicional de se documentar requisitos se dá através da enumeração das funcionalidades e das características que o sistema deve possuir em uma grande lista, formando uma espécie de contrato. Em sistemas de maior complexidade, estas listas costumam formar um grande bloco com centenas de páginas, dificultando o acesso a estas informações que são o principal insumo para a criação da solução.

Outra técnica bastante conhecida para a documentação de requisitos de software é a elaboração de **casos de uso** (HEUMANN, 2008). O caso de uso descreve como serão as interações do usuário final como sistema. Estas interações se dão por meio de atores, que são as representações da comunicação do sistema com o mundo exterior (seja através do usuário, seja através de outros sistemas, seja através de dispositivos de hardware). A descrição de cada interação é dividida em passos seqüenciais, que podem

levar a caminhos alternativos. Podemos ver um modelo de especificação de casos de uso no Anexo 1.

Cada possibilidade de seqüência de passos do início ao fim de um caso de uso recebe o nome de cenário. O resultado final de uma especificação de caso de uso é um texto encadeado logicamente que traz uma linguagem menos técnica com a boa intenção de ser entendido tanto pelo usuário final quanto pelos desenvolvedores.

Os casos de uso se apóiam em uma representação em forma gráfica. Esta representação, denominada **diagrama de casos de uso** focaliza o aspecto de informar genericamente os atores e a relação de um caso de uso com outros casos de uso, uma vez que existe a possibilidade dos casos de uso se referenciarem entre si. Não é o objetivo desta representação mostrar o fluxo de eventos que ocorre durante as interações do usuário com o sistema.

Casos de uso ainda precisam ser verificados e validados. A validação de um caso de uso baseia-se da definição de *critérios de aceite*, que podem ser escritos no documento de especificação do caso de uso. Estes critérios serão exaustivamente testados na etapa de testes de software, juntamente com o projetista de testes, eliminando a maior quantidade possível de defeitos antes da entrega para a validação final, junto ao cliente.

3. Verificação e validação de requisitos

Com os requisitos do sistema documentados em uma linguagem compreensível tanto para o cliente quanto para o desenvolvedor, é possível a realização da atividade de verificação. Neste momento, são realizadas revisões nos documentos gerados, em busca de conflitos, eliminando-se defeitos e omissões.

No processo de validação, desenvolvedores enviam a documentação gerada para o cliente. O cliente, registra suas observações e devolve os documentos para o desenvolvimento. Este ciclo se repete até que todas as inconsistências sejam eliminadas.

Na forma tradicional, as verificações e validações de requisitos são tarefas realizada de maneira completamente manual. Ao fim da validação, o cliente registra o seu aceite sobre a documentação gerada, firmando um contrato entre as partes.

A cobertura de todos os requisitos nas revisões de verificação e validação é fundamental, pois assegura o aumento na a qualidade do produto. Caso

algum requisito passe despercebido para uma etapa seguinte, os problemas se propagarão para todas as etapas posteriores, aumentando a quantidade de erros no sistema.

4. Gerenciamento de requisitos e rastreabilidade

Uma das grandes certezas encontradas no ramo de desenvolvimento de software é que tudo que foi acordado em etapas anteriores está passível de mudanças. Mesmo com toda a busca pelo entendimento do domínio de negócio, o inevitável muitas vezes ocorre, seja por forças externas ao projeto seja por evoluções naturais na forma de se repensar o modelo de negócio.

A atividade de gerenciamento de requisitos requer habilidade das partes para renegociação de prioridades, esforço e prazo anteriormente firmados quando uma alteração ocorre. O mais importante desta atividade está em "abraçar" as mudanças, visto que elas são na verdade novas necessidades do cliente, o principal interessado no software em desenvolvimento. As mudanças ocorrem por uma série de fatores como:

- Sistemas complexos dificilmente são previsíveis ao ponto de serem levantados completamente;
- Usuários mudam de idéia, podem realizar tentativas até ajustar o sistema ao que realmente deseja;
- O ambiente externo muda, independendo da vontade do usuário;
- No momento em que o levantamento foi realizado, algumas perguntas deixaram de ser feitas, estas omissões serão fatalmente reveladas no futuro.

O importante é que tanto o gerente de projetos, quanto a equipe de desenvolvimento quanto o cliente assimilem as transformações que ocorrem durante o ciclo de vida. As mudanças impactam em todo o processo de desenvolvimento e requerem respostas rápidas para adaptações ao novo cenário. Neste momento é cabível a realização de uma análise de impacto, fazendo um levantamento de quanto será custoso a incorporação das mudanças no sistema.

As mudanças nos requisitos influenciarão em todas as atividades que já foram realizadas, necessitando de uma atuação sistemática para orientar a incorporação das alterações em um produto cujo desenvolvimento encontra-

se em curso. Dentre estas atividades, temos a realização de coletas e geração de métricas que permitam acompanhar as mudanças estatísticamente. As métricas fornecem levantamentos que ajudam responder às seguintes questões:

- Quantos casos de uso o sistema possui?
- Quais características críticas do sistema foram aprovadas?
- Que documentação precisará ser atualizada?
- Qual o custo estimado das mudanças propostas?
- Qual o número de mudanças que ocorreram no período?
- Quem autorizou as mudanças?
- Qual o impacto destas mudanças em outras partes do sistema?

A *rastreabilidade* é a capacidade de se determinar a correlação de um elemento do sistema com todos outros elementos existentes. Um elemento pode ser uma documentação de requisito do sistema, itens presentes em modelos de análise e projeto, código fonte, documentos gerados para realização de testes, documentação do usuário.

Por estabelecer uma relação entre todas as etapas do desenvolvimento do software, a determinação da *rastreabilidade* é relevante para levantamentos de análise de impacto e garantia da consistência do produto final.

Ao mapear a *rastreabilidade* entre os elementos do sistema, é possível percorrer todo caminho desde a solicitação do usuário até o produto final, determinando em cascata que alterações foram realizadas em cada elemento do sistema. Realizando este levantamento, é possível garantir que todos os requisitos solicitados foram contemplados na implementação, verificando se o aplicativo realiza somente aquilo que foi solicitado que ele fizesse.

No caso de alterações no sistema, tal levantamento pode definir muito bem que modificações precisam ser feitas em todo o conjunto, permitindo mensurar os esforços necessários e os impactos gerados em elementos preexistentes.

Problemas

Após a apresentação da *engenharia de requisitos*, iremos enumerar uma série de questões que tornam esta disciplina uma das áreas mais desafiadoras dentro do processo de desenvolvimento software. Cada técnica

empregada possui uma série de qualidades e defeitos. Assim, mesmo com a utilização das melhores técnicas e um roteiro bem definido, é na realização das atividades que se revelam os maiores obstáculos.

Levantamento de requisitos: o difícil diálogo

O levantamento de requisitos é uma atividade com a participação intensa do analista de requisitos e da equipe de desenvolvimento, juntamente com a área de negócio, cliente e todas as partes interessadas. Nesta fase, os requisitos são catalogados em documentos que seguem uma padronização adotada pela organização.

A comunicação entre pessoas que atuam em áreas tão distintas é difícil, fazendo com que a adoção de padrões de documentação e diagramas ilustrativos nem sempre surtam o efeito desejado. Muitas vezes a equipe de desenvolvimento necessita fixar parâmetros mais técnicos, enquanto a área cliente juntamente com a área de negócio domina o seu fluxo de negócio e as normas que guiam o trabalho da organização.

Este conflito de visões não interessa a nenhum dos participantes. Enumerando alguns problemas comuns, chega-se com facilidade à seguinte lista:

- Os desenvolvedores não entendem o domínio do negócio;
- O vocabulário utilizado pelo cliente não é o mesmo utilizado pelo desenvolvedor, levando a problemas de comunicação;
- O processo do cliente pode não estar devidamente mapeado pela área de negócio;
- Muitas vezes o cliente não sabe expressar de forma clara o que deseja;
- O usuário não tem idéia do que é possível ser feito com as tecnologias existentes;
- Dificilmente o usuário terá condições de responder a perguntas técnicas triviais para o desenvolvedor; para ele é mais fácil dizer em linhas gerais o que ele precisa, ou seja, qual a sua necessidade.

O levantamento dos requisitos ganha importância por ser o primeiro contato que o desenvolvimento tem com o domínio do problema. Um levantamento bem feito elimina uma série de problemas futuros e anula fatores de risco relacionados com o entendimento das necessidades do cliente,

fornecendo para as equipes um fundamento sólido para a construção de um produto tão próximo quanto o possível do que foi pedido. Sob o ponto de vista econômico, inúmeros estudos convergem para a conclusão de que o retorno obtido com a eliminação de defeitos ainda na fase inicial de levantamento de requisitos diminui o custo total do projeto (AVILA, 2008). Em resumo, é mais barato investir em corrigir problemas nos requisitos do que nas fases subseqüentes do projeto.

Uma ilustração muito bem humorada (GOMES, 2008) reforça ainda mais as afirmações feitas acerca importância da atividade inicial de levantamento de requisitos. As imagens retratam uma realidade bastante comum, encontrada em projetos que, por conta de uma atividade anterior elaborada com problemas, acaba por gerar um produto final muito diferente da real necessidade do cliente, trazendo um desfecho ruim tanto para o cliente quanto para todos envolvidos.

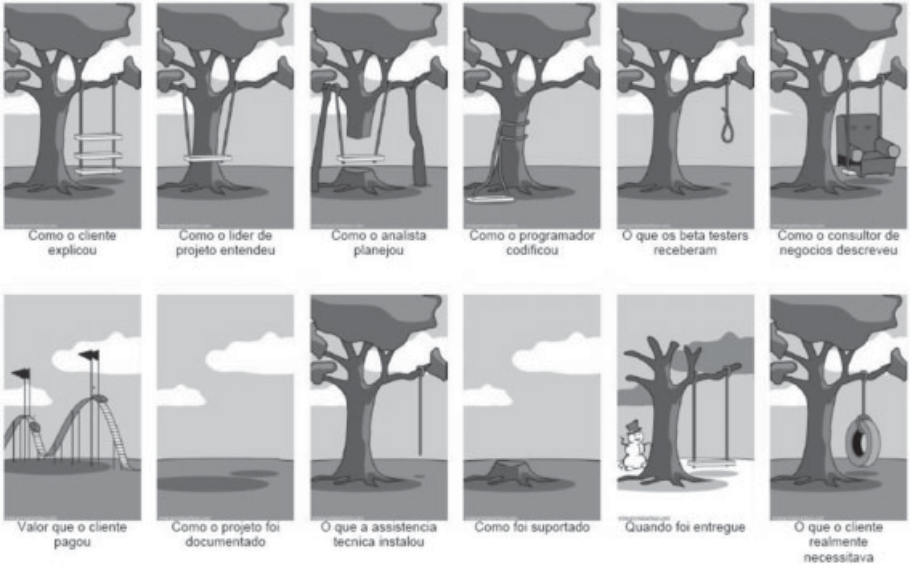


Figura 1. Ilustrando os problemas nas etapas de desenvolvimento de software

Em todas etapas da engenharia de requisitos, a comunicação entre as partes é um fator crítico. A falta de entendimento entre os envolvidos que possuem visões distintas do mesmo problema é um fato comum. O estado da

arte nesta fase, seria disponibilizar um canal de comunicação em que todos os envolvidos pudessem descrever de forma gráfica ou textual uma visão do problema ou um protótipo de telas. Ao fim deste processo, bastaria mapear os pontos de concordância e chegar a um consenso sobre os pontos de divergência.

Documentos eletrônicos e suas limitações

O termo "documentar requisitos" remete ao tempo do uso de "papéis datilografados" para registro das funcionalidades de um sistema que até hoje se mantém, ainda que na versão de "documentos eletrônicos". Percebemos, no entanto, que o armazenamento em "documentos eletrônicos" (Anexo 1), mesmo representando um avanço sobre os "papéis datilografados", ainda possui grandes limitações, como por exemplo:

(a) Os requisitos possuem correlações entre si. Representar estas ligações em documentos eletrônicos é custoso, consome muito tempo das equipes. É uma tarefa trabalhosa e propensa a muitos erros;
(b) É difícil manter, manualmente, em forma de documentos eletrônicos, um histórico das alterações que foram realizadas e por quem foram realizadas;
(c) Os requisitos possuem atributos e estados. É possível, porém custoso representar estas informações em documentos "planos". Mesmo utilizando bases de dados das ferramentas de gerenciamento de requisitos, muitas informações referentes a atributos e estados poderiam ser preenchidas automaticamente, e não manualmente, gerando erros. Como exemplos de atributos, os requisitos podem ser "funcionalidades", "restrições tecnológicas", podem ter prioridade "alta, média ou baixa". Como exemplos de estados, os requisitos podem estar em situação "proposta, em elaboração, em revisão, aprovado, implementado, testado, implantado";
(d) Realizar o cruzamento de informações em documentos é uma tarefa custosa;
(e) Documentos eletrônicos possuem um formato próprio, o que traz uma dificuldade adicional quando a organização resolve adotar um formato diferente de documento, por questões de incompatibilidade. Nem todos os softwares editores de documentos geram formatos

compatíveis entre si, necessitando de ajustes para resolver problemas de compatibilidade;

(f) Documentos desformatam-se facilmente, bastando em muitos casos a utilização de um software leitor de documento em uma versão diferente do software que gerou o documento. Este problema é mais difícil de ocorrer quando a organização utiliza-se de padrões abertos para a produção de seus documentos;

(g) Documentos eletrônicos podem possuir erros de digitação que poderiam ser facilmente evitados. Citemos alguns erros comuns: data de criação do documento incoerente, omissão do autor do documento, omissão do preenchimento de campos obrigatórios do documento, etc. Erros que poderiam ter sido colhidos através de consultas a outros sistemas o terem seu preenchimento automatizado. Analisando o Anexo 1, podemos levantar outros campos importantes cujo preenchimento falho comprometeria o entendimento do conteúdo;

(h) Um mesmo documento pode ser utilizado por pessoas de áreas completamente diferentes. Neste caso, os detalhes do documento para pessoas de uma área técnica não interessam para pessoas de uma área gerencial ou de negócio;

(i) Em configurações usuais, documentos não foram criados para serem acessados e alterados simultaneamente.

Verificando e validando casos de uso

Vimos anteriormente que as especificações de casos de uso são documentos textuais que descrevem em linguagem simples as interações do usuário com o sistema. Apesar desta aparente simplicidade, a representação dos casos de uso em documentos textuais ou em diagramas de casos de uso limita a compreensão do sistema em diversos aspectos. A equipe pode contar com a especificação de casos de uso como uma ferramenta auxiliar, mas incompleta para a documentação de requisitos cujo principal objetivo é registrar fragmentos do sistema, permitindo compreendê-lo em sua totalidade.

Um problema encontrado com a elaboração dos casos de uso está na abordagem textual. A validação de longos textos é uma tarefa enfadonha e cansativa. Muitas vezes os clientes não compreendem os textos elaborados, não conhecem a técnica de casos de uso ou não têm condições de, sozinhos,

analisarem a documentação enviada para verificação. Durante a revisão do texto, pode ser necessário registrar pequenas anotações, tirar dúvidas ou realizar correções no próprio documento. Para isto, é preciso auxílio de um software que mantenha o histórico destas alterações, permitindo comparações posteriores.

Uma possível representação de caso de uso pode ser feita representando-se o fluxo de forma visual em vez de utilizar a forma textual. Neste caso, o apelo visual é bem maior. A apresentação da seqüência de eventos que ocorre em um sistema pode feita por meio de fluxos gráficos com conectores, representando as condições e rumos que o fluxo pode tomar de acordo com as restrições impostas. Isto pode ser uma boa saída para amenizar o problema gerado por textos extensos.

Outro problema percebido está na falta de uma ferramenta que guie os participantes na realização do fluxo completo das atividades de revisão. Por este motivo, requisitos podem passar despercebidos, sem sofrerem um processo de verificação ou validação, causando impactos nas fases futuras do ciclo de vida como um todo.

No contexto dos requisitos, o envolvimento de todos na verificação é fundamental para que se tenha uma visibilidade completa da atual situação do projeto. Sem a automatização do fluxo, o processo é realizado através do o envio de documentos para o cliente e do aguardo do aceite sobre determinado documento. Ferramentas para documentação de requisitos apresentam uma interface pobre, não encorajando a participação e de todos neste processo de inspeção.

O esforço para se manter a rastreabilidade de requisitos

O mapeamento entre os documentos do sistema é um ponto crucial na determinação da rastreabilidade. Ferramentas de documentação de requisitos oferecem funcionalidades para mapeamento de rastreabilidade, favorecendo a realização da análise de impacto. Nestas ferramentas, o usuário informa manualmente a navegação entre os requisitos, acessando uma interface conhecida como *matriz de rastreabilidade*.

Em casos de mudanças em requisitos, a tarefa de manutenção da rastreabilidade torna-se um grande fardo. A alteração em um software implica em verificar constantemente as relações de dependência entre os requisitos, além da atualização em toda a documentação.

O levantamento do impacto nas alterações de requisitos também é feito manualmente. Por isso, consome tempo da equipe envolvida no processo. Se as informações necessárias estivessem armazenadas adequadamente em bases de dados, este levantamento poderia ser feito de forma automatizada.

Problemas Internos

Hoje, a empresa tem dois grandes grupos de desenvolvedores: os que trabalham com software livre e os que trabalham com softwares proprietários. O segundo grupo, dos softwares proprietários, na sua grande maioria, trabalham com sistemas legados que foram construídos na empresa quando ela ainda trabalhava com esse tipo de software e filosofia.

Atualmente, dentro da empresa e no governo, é uma tendência o movimento para software livre, seja ele no uso de ferramentas de apoio ao desenvolvimento ou de ferramentas de monitoração e acompanhamento de serviços ou mesmo de software básico. Um grande exemplo disso foi a migração de quase toda a base de empregados do SERPRO para o Fedora Linux como sistema operacional e o BROffice como suíte de escritório (texto, planilhas, apresentação, etc).

Como existe uma grande base de desenvolvedores atuando com sistemas totalmente em software livre, havia um grande problema: quando se trabalhava com requisitos, o analista de requisitos tinha que mudar da plataforma Linux para Windows apenas para utilizar o RequisitePro. Além de consumir um pouco de tempo nesta tarefa, que ao total do projeto pode somar um tempo considerável, obrigava o desenvolvedor a ter o Windows na máquina, mesmo com uma resolução de se usar software livre na empresa.

Isso obrigava a empresa a arcar com um custo aproximado de R$2,4 milhões, calculados da seguinte forma:

- Requisite Pro é cerca de 1.7 milhões de reais, calculados da seguinte forma:

  licença do RequisitePro Floating user: R$8.900,35 (TRTRIO, 2006)(BNB, 2003)

  quantidade de licenças: 200 licenças, onde apenas 200 usuários conseguem utilizar a ferramenta simultaneamente

  custo do suporte de 12 meses: 2.965,25

  custo total estimado: 8.900,35 x 200 + 2.965,25 = 1.783.035,25

• Para a documentação de sistemas (utilizando-se do MS Word) em cerca de 620 mil reais assim calculados:

custo total estimado da licença de 200 reais x 3.100 usuários entre desenvolvedores, gerentes, clientes e analistas de negócio.

Além disso, a ferramenta tem alguns problemas que não a tornam intuitiva ou de fácil utilização: ela não separa a informação que deseja guardar e/ou recuperar da apresentação; não permite troca de formato da apresentação; não notifica que houve problemas; entre outros.

Isto leva o usuário a um estudo da ferramenta demasiado longo ao entrar na empresa. Este estudo tem formato de curso com quarenta horas de duração. Uma ferramenta intuitiva, com boa usabilidade e que refletisse o fluxo utilizado no processo de requisitos.

Apresentando o RekZit

Mostraremos a seguir uma série de soluções para os problemas levantados anteriormente. Parte destas soluções encontram-se implementadas na ferramenta denominada RekZit, que é o objeto principal deste artigo. Outra parte, está sugerida como propostas de funcionalidades futuras na ferramenta. Serão também mostradas algumas telas do sistema, de forma a ilustrar o trabalho já realizado.

Características gerais da solução

O RekZit encontra-se disponível em forma de protótipo funcional, no endereço http://10.200.107.38:30, acessível somente pela rede interna do SERPRO. Para conhecer melhor a ferramenta, é possível entrar em contato com a comunidade de desenvolvedores da através de um sistema de colaboração acessível no endereço http://colab.serpro, também através da rede interna. Embora a principal interface de utilização da ferramenta seja pela *web*, existem ainda possibilidades se implementar outras interfaces. Uma delas em forma de um barramento de serviços, servindo de catálogo de informações, porta para integrações futuras.

A solução RekZit foi implementada no estilo *web 2.0*[7], possibilitando maior interatividade com o usuário, permitindo simular a experiência de

[7] Web 2.0 - Expressão usada para se referir a uma segunda geração de serviços e comunidades encontrados na *web*, favorecendo sobretudo a interatividade com o usuário e interfaces gráficas ricas.

aplicações executadas localmente no *desktop*. A linguagem de programação adotada foi o PHP. Linguagem de desenvolvimento Livre, que permite a orientação a objetos, com o principal propósito de implementar sistemas *web* seguros, eficientes e velozes. Além destas vantagens, PHP é notável pela sua ótima curva de aprendizado possuindo uma vasta documentação, inúmeras bibliotecas de funções e *frameworks* de desenvolvimento largamente utilizados, como é o caso do Symfony[8] e do Zend Framework[9]. Como exemplo de utilização de PHP em larga escala, temos os sistemas da Yahoo (LERDORF, 2008) e o da Wikipedia[10], este último conhecido como uma enciclopédia livre, com grande volume de acessos.

O sistema implementado segue a padronização W3C[11], que garante a execução em qualquer navegador *web*. Por ser uma aplicação que pode ser executada na *web*, em qualquer browser, a solução afasta definitivamente qualquer problema de incompatibilidade com outras plataformas e dificuldades na instalação. Com o RekZit, qualquer usuário autorizado que esteja conectado à rede interna do SERPRO poderá, a qualquer momento, acessar a documentação dos sistemas, sem a necessidade de ter uma instalação da ferramenta proprietária RequisitePro.

O RekZit, foi concebido como uma ferramenta multi-sistemas e multi-projetos. Na implementação proposta, é possível navegar entre diversos sistemas da organização. Em cada sistema, é possível selecionar um dos projetos de software.

Os requisitos de software estão associados tanto a nível de sistema quanto a nível de projeto. Isto dá a seguinte flexibilidade:

1. Nível de sistema: é possível listar todos requisitos do sistema selecionado;
2. Nível de projeto: é possível listar requisitos que estão sendo alterados em determinado projeto.

[8] Symfony - http://www.symfony-project.org/ *framework* livre para desenvolvimento rápido de sistemas em PHP

[9] Zend Framework – http://framework.zend.com/ *framework* para desenvolvimento de sistemas em PHP criado pela Zend, grande patrocinadora do PHP

[10] Wikipedia, a enciclopédia livre, disponível em http://pt.wikipedia.org/wiki/P%C3%A1gina_principal

[11] Mais informações em: http://www.w3.org/

Este tipo de associação permite realizar uma *rastreabilidade* automática, enumerando os requisitos que foram desenvolvidos em determinado projeto, ou ainda listar todos os projetos que alteraram determinado requisito. Ainda em termos de documentação, a solução elimina a necessidade de armazenar informações redundantes em artefatos de sistema e artefatos de projeto. Isto ocorre no *documento de visão do sistema* e *documento de visão do projeto*, um dos documentos padrões adotados no PSDS. No Anexo 2, podemos visualizar uma tela do RekZit, mostrando os projetos associados a um sistema.

Levantando glossário do sistema com esforço mínimo

Capturar a linguagem utilizada pelo cliente é uma tarefa essencial no levantamento de requisitos. Esta atividade deve ocorrer da maneira mais direta e mais cedo possível. Uma boa prática adotada na engenharia de requisitos é a elaboração de uma relação dos termos utilizados no domínio do negócio, juntamente com o significado de cada termo para o sistema em questão formando um *glossário do sistema*. No desenvolvimento do sistema, é natural que estes termos passem por sucessivas revisões, refinando os seus significados, gerando uma lista cada vez mais consistente.

Na ferramenta proposta, esta necessidade encontra-se implementada com um cadastro simples de termos, que pode ser submetido a uma busca textual em uma base de dados. A busca também pode ser feita por palavras correlatas ao termo buscado. Atualmente a inclusão de novos termos pode ser feita por qualquer pessoa envolvida no projeto, recomendando a participação do cliente. A validação de um termo, por sua vez, é tarefa exclusiva do cliente juntamente com a área de negócio.

Por estar disponível a todos, a coleta de termos pode ser feita de forma colaborativa, com a participação simultânea das partes interessadas. Na medida em que toma contato com o domínio do negócio, a equipe de desenvolvimento pode acrescentar termos ao glossário, cujos significados poderão ser validados posteriormente pelo cliente.

A utilização deste cadastro não se limita a uma simples recuperação do termo para acesso ao seu significado. Indo mais além, foi implementada uma funcionalidade para ligar os termos aos requisitos do sistema. Os termos que fazem parte deste glossário podem ser citados em qualquer

documento que faça parte do sistema, tornando-se um *link* de acesso direto ao significado do termo utilizado.

É possível melhorar esta funcionalidade, implementando uma busca em todos os documentos do sistema que fazem menção a um determinado termo citado. Desta forma, pode-se afirmar que o sistema possibilita definir uma rastreabilidade automática dos requisitos através dos termos utilizados, com ganhos de tempo, aumento de consistência, eliminação de redundâncias e redução no custo total da manutenção de documentos de requisitos.

Documentando casos de uso com uma interface melhorada

Dentre as dificuldades e limitações levantadas, concluímos que o armazenamento das documentações de requisitos não deva ser feito por meio de documentos eletrônicos, mas sim por uma base de dados compartilhada. Esta base de dados pode ser acessada simultaneamente por vários usuários, através de uma aplicação principal. Isto garante que os requisitos documentados estejam disponibilizados para todos os envolvidos no projeto, oferecendo vantagens sobre o armazenamento em forma de documentos eletrônicos. No Anexo 3, podemos ver o aspecto de uma especificação de caso de uso no RekZit.

O fluxo de eventos do caso de uso também pode ser melhor representado por meio de fluxogramas simples. O maior ganho com esta abordagem está na possibilidade de se representar a mesma informação do caso de uso, acrescentando formas visuais que facilitam a compreensão. Ao separar a representação da informação, é possível também implementar uma série de artifícios como por exemplo, registrar detalhes que seriam interessantes para a área técnica, sem que os membros da área de negócio tenham o entendimento prejudicado. No Anexo 4, podemos visualizar o aspecto da representação dos fluxos de um caso de uso na forma visual.

Futuramente, esta mesma base poderá ser alimentada por outras ferramentas de desenvolvimento, como por exemplo, as ferramentas para solicitação de demandas do cliente (área de negócio), ferramentas responsáveis pela modelagem da solução (análise e projeto da solução tecnológica), ferramentas de implementação da solução propriamente dita, ferramentas de testes de aceitação e finalmente chegando ao produto final.

Verificação de requisitos *online*

A atividade de verificação e validação de requisitos aqui apresentada possui uma série de dificuldades, relacionadas principalmente com a falta de interação entre desenvolvedores e usuários finais. Em termos gerais, esta atividade deveria ser realizada de forma mais colaborativa.

A automatização do fluxo de envio de documentos, registro de considerações do cliente e eliminação de inconsistências até chegar ao documento final, facilita a cooperação entre as equipes, uma característica importante da tarefa de validação. Esta característica deve ser valorizada, em contraste com a realização de uma tarefa cansativa e solitária, como ocorre no modelo tradicional.

Uma questão problemática na validação está na dificuldade de comunicação devido à distância entre a equipe de desenvolvimento e o cliente. Esta dificuldade causa um atraso na chegada de informações, sobretudo na avaliação dos artefatos enviados para validação. Por conta deste atraso, os projetos tornam-se morosos colocando em risco o prazo acordado.

A validação *online* é possível com a adoção do RekZit, que permite o envolvimento da área cliente com o analista de requisitos. O envolvimento de todos pode ser feito de forma simples, através de notificações enviadas por *e-mail* no momento em que o cliente valide o requisito.

Com isto, todos podem acompanhar a situação mais atual dos requisitos, no instante em que acessam a ferramenta. Requisitos validados e verificados *online* podem passar imediatamente para a fase seguinte do desenvolvimento, agilizando os projetos e otimizando o tempo.

Dos requisitos para a análise e o projeto

As fases subseqüentes da engenharia de software, após a fase de requisitos, abrangem uma série de análises realizadas sobre a documentação gerada, passando pelo domínio do problema para a concepção da solução técnica. Nestas fases posteriores, a integração com ferramentas de desenvolvimento para a análise e projeto de sistemas é importante.

Uma ferramenta de requisitos integrada com as fases de análise e projeto, possibilita ao desenvolvedor fazer mapeamentos do *caso de uso* ao *modelo de análise*, que por sua vez será mapeado ao *modelo de*

projeto, até o *código fonte*. O ganho obtido com este mapeamento possibilita o rastreamento de requisitos até o código fonte, uma das metas apontadas no modelo CMMI.

Dos requisitos para as validações

Para cada requisito do sistema, é possível estabelecer um critério de aceitação. Este critério é uma condição que precisa ser atendida para que este requisito seja considerado válido. Um caso de uso ou uma história do usuário pode ser descrita de forma que seus passos possam ser validados pelo usuário. Os critérios de aceitação são, em resumo, uma linha de fronteira entre a fase de requisitos e a fase de testes.

Um caso de uso implementado e passado em todos os critérios de aceite significa que o desenvolvimento está concluído, podendo passar para uma fase posterior. Na ferramenta proposta, é possível gerar critérios de aceitação em diversos pontos da documentação, facilitando uma melhor integração com os testes. Itens de requisitos tais como, casos de uso, passos de caso de uso, e até mesmo regras de negócio podem ser associados a um critério de aceite. Estes critérios de aceite por sua vez podem ser associados a casos de testes, permitindo uma rastreabilidade do caso de uso ao caso de teste.

Esta associação abrirá portas para a realização das atividades de testes de validação. Um caso de uso que estiver no status "implementado", poderá disparar uma solicitação para o projetista de testes, sinalizando que aquela parte do sistema encontra-se disponível para ser testada.

Trabalhos Futuros

Uma ferramenta é produtiva quando traz ganhos logísticos às atividades envolvidas ou quando facilita o trabalho automatizando algumas funcionalidades que eram feitas manualmente pelos especialistas na atividade. Dito isto, alguns dos trabalhos futuros dentro da ferramenta seria automatizar atividades que são repetitivas e que podem ser conseguidas com cálculos ou ativação de processos.

A rastreabilidade automática permitirá conhecer impactos imediatamente após a modificação. Essa mesma rastreabilidade poderá gerar um relatório de análise de impacto automático após as modificações provisórias nos requisitos. O cliente poderá receber um notificação quando os requisitos

estiverem prontos para validação. O líder de projeto poderá receber uma notificação quando os requisitos estiverem verificados. Algumas interações entre requisitos serão automáticas, feitas no momento de suas inclusões no sistema.

Outros trabalhos futuros interessantes seriam as integrações com outros sistemas. Não estamos falando aqui de integrações óbvias e que precisariam ser feitas de imediato, como a integração com o SGI (Sistema de Gestão de Informações), mas sim com ferramentas que auxiliariam em outras etapas do processo de desenvolvimento de software.

Estudos vêm sendo feitos no SERPRO para a internalização de algumas ferramentas de engenharia de software baseadas em softwares livres. Entre elas, ferramentas de análise e projeto e de implementação. Outros estudos estão sendo feitos com relação a outras macroatividades, como testes e homologação. O MOSKitt (MOSKitt, 2008) foi apresentado ao SERPRO como alternativa ao Rational Rose. O TestLink (TESTLINK, 2008), como alternativa ao Rational Test Manager. Está em constante desenvolvimento o Framework de Desenvolvimento Java do SERPRO (FRAMEWORK, 2008).

Estas ferramentas poderiam ser integradas ao RekZit para buscar informações. O MOSKitt poderia buscar informações sobre casos de uso para executar sua realização. O TestLink poderia buscar informações sobre os critérios de aceite para fomentar os casos de teste. O Framework SERPRO poderia se valer dessas informações para geração de código e geração de scripts de teste.

Este último ponto levanta uma interessante questão: o desenvolvimento integrado de ponta-a-ponta do software utilizando-se apenas de softwares livre. O RekZit abre uma oportunidade de desenvolver uma solução completa dentro da empresa e colocar a disposição da comunidade de software livre.

Esta solução utilizaria as informações armazenadas no RekZit com todos os registros de mudanças, apresentando as informações da maneira que for mais conveniente ao cliente e preparando estes requisitos para a próxima etapa. Estes dados seriam tratados e utilizados pelo MOSKitt para realizar os casos de uso, tratar as regras de negócio ou mesmo se preparar para os requisitos não funcionais. O Framework SERPRO se utilizaria destes dados gerados no MOSKitt para geração de código automático, deixando para o implementador se preocupar apenas com regras mais específicas do negócio. O TestLink, através dos critérios de aceite do RekZit, criaria alguns casos de teste que poderiam ser usados como insumo na geração de scripts de

execução, que seriam utilizados no Framework SERPRO dentro da integração contínua.

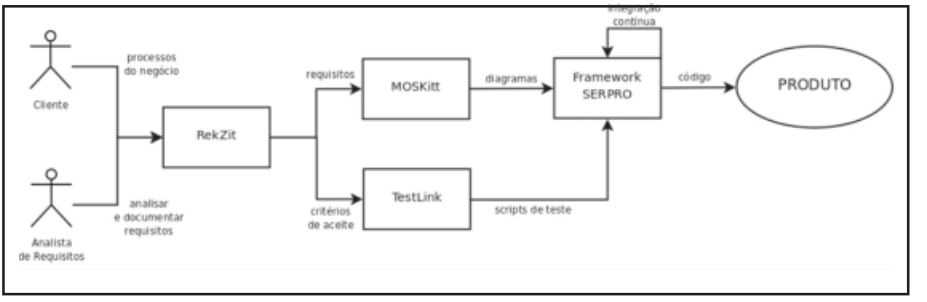


Figura 2. Ilustração da integração entre as ferramentas

Esta foi apenas uma ilustração dessa integração que faria parte de uma solução completa para desenvolvimento de software, contando apenas com ferramentas que se tem notícia dentro da empresa (Figura 2). Se contarmos com idéias que estão surgindo, poderia se integrar ainda com ferramentas de prototipação ou mesmo disponibilizar serviços no futuro barramento corporativo de serviços (ESB – enterprise service bus) do SERPRO.

Finalmente, o trabalho futuro mais importante no momento é transformar este protótipo em um sistema oficial do SERPRO.

Conclusão

Uma ferramenta deste porte dentro do SERPRO pode trazer grandes ganhos para empresa. Ganhos financeiros, com a redução de custos pelas licenças de RequisitePro, MS Word e Windows não renovadas. Ganhos de processo, com uma ferramenta desenvolvida internamente, refletindo a macroatividade de requisitos do PSDS. Ganhos logísticos, com automatização de tarefas repetitivas, como a extração imediata de indicadores, ou mesmo a integração com outros sistemas, não criando informações redudantes.

Ainda traz outros ganhos de difiícil quantificação. Não existe, no mercado, uma ferramenta livre de requisitos que seja utilizada em larga escala. Existem algumas, mas estas não atendem às necessidades de grandes empresas de tecnologia. Se disponibilizado o código fonte desta ferramenta nas comunidade

livres, os ganhos com a disseminação e compartilhamento do conhecimento trarão um grande destaque para a empresa. O uso da ferramenta poderia ser estendido a outras unidades do governo, dando ganhos financeiros a estas unidades e trazendo novos parceiros para o leque de clientes do SERPRO. Tendo em vista que o SERPRO está caminhando na direção do software livre, esta é uma grande oportunidade de negócio.

Como resultado, podemos mostrar que o acompanhamento de alguns requisitos foram feitos utilizando a ferramenta, como por exemplo, a montagem do glossário de alguns sistemas. Esta etapa se mostrou bastante eficiente, com a participação direta do cliente na descrição de alguns termos. Isso facilitou o trabalho e diminuiu o esforço dos analistas de requisitos no projeto, que não precisaram confirmar a descrição dos termos do glossário.

A verificação dos requisitos, o acompanhamento do trabalho dos analistas e o controle de recursos ficou facilitado. Todas estas atividades foram reunidas em uma ferramenta de acompanhamento *web*, onde ficou evidente o trabalho de cada um dos analistas. Uma prova de conceito mais completa está sendo preparada para outras etapas do processo, assim que a funcionalidade estiver disponível no protótipo.

ANEXOS

Anexo 1: Documento modelo de uma especificação de caso de uso em forma de documento texto

<Nome do Caso de Uso> <opcional>

Histórico de Versões

Data	Versão	Descrição	Autor	Revisor	Aprovado por
<dd/mm/aaaa>	<x.x>	<detalhes>	<AREQ>	<AREQ ou LP>	<LP ou CLI>

Especificação de Caso de Uso

· **Nome do Caso de Uso**

<Iniciar com verbo no Infinitivo. Utilizar nome completo, onde as primeiras letras devem ser maiúsculas, exceto preposições. Este nome deve retratar claramente a ação a ser realizada. >

· **Objetivo**

<Descrição da finalidade do caso de uso. Deve oferecer uma idéia geral do propósito do caso de uso.>

· **Tipo de Caso de Uso** <opcional>

<Classificar o Caso de Uso em questão em Concreto ou Abstrato. Um caso de uso concreto é iniciado por um ator e constitui um fluxo completo de eventos. "Concreto" significa que a instância do caso de uso executa o fluxo inteiro invocado pelo ator.

Um caso de uso abstrato nunca é instanciado diretamente por um ator. Casos de uso abstratos são incluídos em (Relacionamento de Include), estendidos por (Relacionamento de Extend), ou especializados em (Relacionamento de Generalização) outros casos de uso.

Quando um caso de uso concreto é iniciado, uma instância do caso de uso é criada. Esta instância também exibe o comportamento especificado pelo caso de uso abstrato associado. Assim, instâncias do caso de uso abstrato não são criadas em separado.

A diferença entre os dois tipos é importante, pois os atores irão "ver" e iniciar os casos de uso concretos no sistema.>

· Atores

Nome Ator	Tipo	
	Primário	Secundário
<Relacionar os atores que interagem com o caso de uso, começando pelo ator que inicia a interação. Utilizar nome completo, onde as primeiras letras devem ser maiúsculas, exceto preposições.>	<Marcar com um X caso o ator inicie o caso de uso.>	<Marcar com um X caso o ator participe forma passiva, somente recebendo informações, por exemplo.>

· Pré-condições <opcional>

<Texto livre que identifica as pré-condições do caso de uso. Pré-condições são condições que devem ser satisfeitas para que o caso de uso possa iniciar. Na inexistência de pré-condições para o caso de uso, formatar a mensagem: **"Nenhuma pré-condição identificada"**.>

· Fluxo Principal

< Descrever o cenário mais utilizado pelos atores. Este cenário apresenta o caminho percorrido pelo ator para atingir o objetivo com sucesso. A numeração dos passos deve ser seqüencial e iniciar em 1. A numeração dos subpassos deve preservar o índice do passo-pai, acrescido ao número identificador do subpasso, iniciando em 1. Caso exista regra de negócio associada ao passo do fluxo, referenciá-la.>

P<n>. <Título do passo>

<Descrição do passo. Sentença clara e objetiva descrevendo a função que o passo declara. Evitar a fragmentação de passos provocada pela decomposição funcional. *Opcional na existência de subpassos ou no caso de título e descrição serem iguais*.>

P<n.m>. <sentença sucinta descrevendo a função que o subpasso declara. A adoção de subpassos é opcional, sendo recomendada apenas para passos complexos.>

· Fluxos Alternativos <opcional>

<Descrever os cenários alternativos utilizados pelos atores. A numeração dos procedimentos deve ser seqüencial e iniciar em 1. Caso exista regra de negócio associada ao passo do fluxo, referenciá-la. Os

caminhos alternativos devem percorrer um fluxo completo, demonstrando:

- o passo a que estão associados.
- a condição que aciona a entrada no caminho alternativo
- as ações tomadas no caminho alternativo
- a ação de retorno. >

A<n>. <descrição do fluxo alternativo.>

· **Fluxos de Exceção** <opcional>

<Descrever os cenários de erros. Estes cenários apresentam os possíveis erros que podem ser observados quando da interação dos atores com a aplicação. A exemplo dos fluxos alternativos, descrever um fluxo completo, observando o disposto nos itens a, b, c, e d anteriores (seção ·ð).>

E<n>. <descrição do fluxo de exceção.>

· **Pós-condições** <opcional>

<Texto livre que identifica as pós-condições. Pós-condições são condições que podem ser garantidas como verdadeiras ao final do caso de uso. Na inexistência de pós-condições para o caso de uso, formatar a mensagem: **"Nenhuma pós-condição identificada"**.>

· **Requisitos Não Funcionais** <opcional>

<Enumerar os requisitos não funcionais identificados especificamente para este caso de uso. Descrever nesta seção requisitos relativos ao caso de uso que não são cobertos pelo fluxo de eventos, ou por outros documentos (Visão, RNF), mas que podem influenciar o desenvolvimento do sistema. Na inexistência de requisitos não funcionais específicos do caso de uso, formatar a mensagem: **"Nenhum requisito não funcional identificado"**.>

· **Ponto de Extensão** <opcional>

<Fazer referência a pontos de extensão utilizados no detalhamento. Pontos de extensão são referências a outros casos de uso externos que complementam o fluxo de eventos do corpo do caso de uso chamador. Identificar o passo do fluxo básico ao qual a extensão está associada, descrevendo as condições e o momento para sua ativação. Na inexistência

de pontos de extensão, formatar a mensagem: **"Nenhum ponto de extensão identificado".>**

PE<n>. <Título do ponto de extensão>
<descrição do ponto de extensão.>

· Critérios de Aceite (cenários de teste)

<Documentar, de forma sucinta, os critérios de aceite ou casos de teste iniciais que possam auxiliar a completa validação do caso de uso para serem utilizados na fase de testes, onde devem ser mais bem detalhados, utilizando a ferramenta de testes. Todos os passos dos fluxos principal, alternativos e de exceção devem estar contemplados nos critérios de aceite. Os critérios de aceite/casos de teste explicitam necessidades de teste e são importantes para validação pelo cliente. >

	Cenário de Teste	Critério de Sucesso	Critério de Falha
CA1	<Ex: Adicionar usuário com dados inválidos>	<Ex: O aplicativo não permite inclusão e devolve mensagem de erro informando onde o problema foi encontrado.>	<Ex: Aplicativo permite inclusão OU o aplicativo apresenta mensagem de erro, mas inclui os dados de qualquer forma.>
CAn			

· Freqüência de Utilização

<Informar se é alta, média ou baixa e quais informações são mais acessadas. A freqüência de utilização deve ser uma estimativa da quantidade de utilização do caso de uso pelos atores em um determinado período de tempo. Caso existam picos de utilização, esses devem ser descritos.>

· Interface Visual <opcional>

<Apresentar o Leiaute das Telas utilizadas neste caso de uso. Neste item, pode-se também definir Regras de Validação e de Formatação, Navegablidade e Ajudas de Contexto. Caso seja necessário utilizar um artefato em separado, indicar seu nome.>

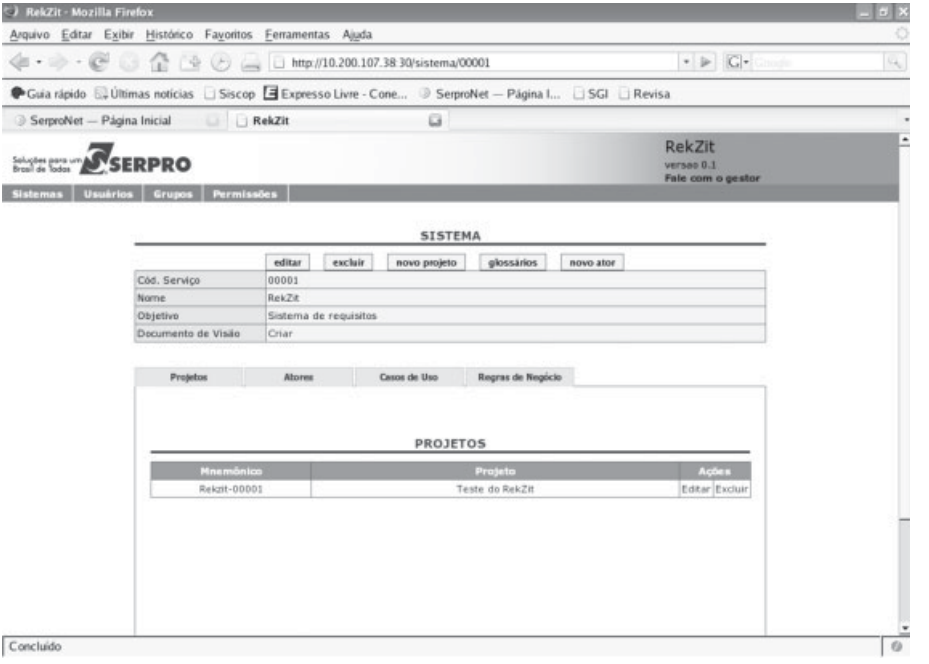
· Observações <opcional>

<Espaço destinado para as anotações técnicas, informações adicionais a serem trabalhas no futuro, ou lembretes.>

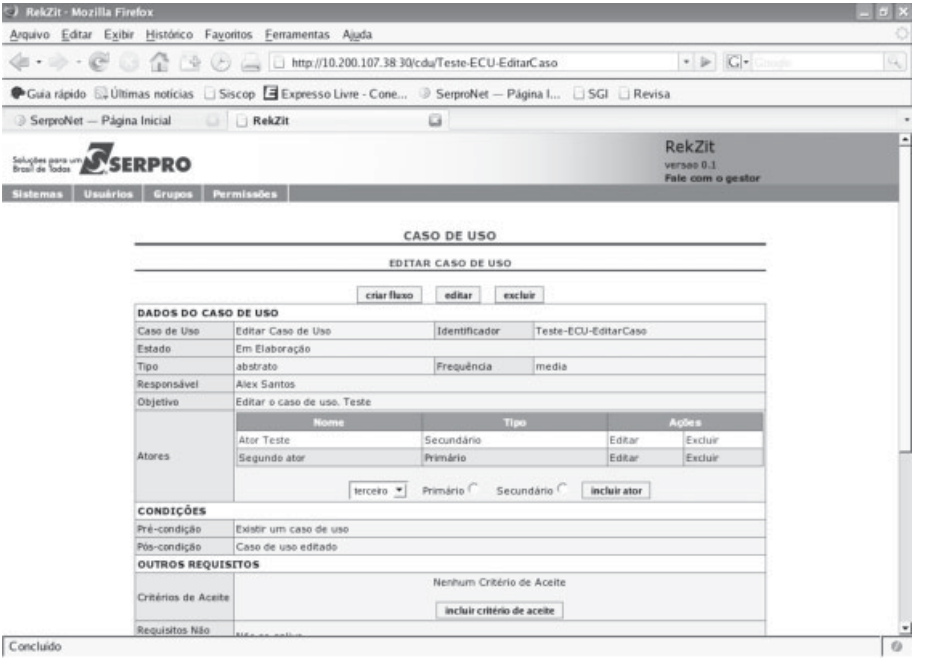
· **Referências** <opcional>

<Item disponível para citar os modelos, diagramas, funcionalidades de outros projetos, e outros documentos que se relacionem ao caso de uso em questão.>

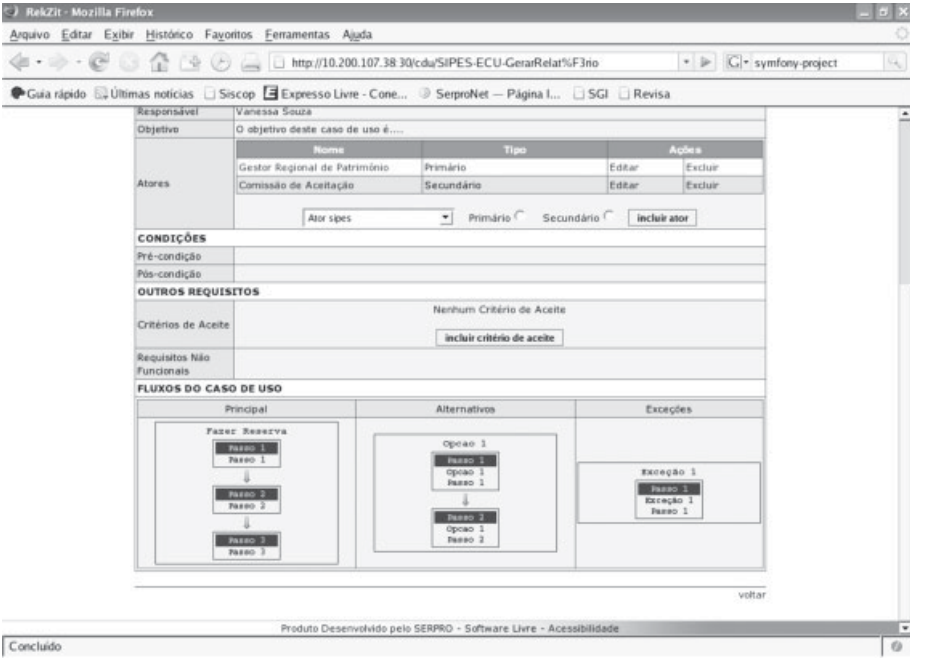
Anexo 2: Tela do RekZit: listagem dos projetos de um sistema



Anexo 3: Tela do RekZit: caso de uso



Anexo 4: Exemplo de uma especificação de caso de uso no RekZit, com fluxo gráfico



Bibliografia

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML**: Guia do Usuário.1ª ed. Rio de Janeiro: Campus, 2000.

SOMMERVILLE, I. Software Engineering. 6a ed. Addison-Wesley Publishing Company, 2001.

SEI - Software Engineering Institute. Capability Maturity Model Integration (CMMI SM), Version 1.2. Relatório Técnico, report CMU/SEI-2002-TR-011, Software Engineering Institute (SEI), 2002.

AVILA, L. Ana - Artigo Engenharia de Software - Introdução à Engenharia de Requisitos http://www.devmedia.com.br/articles/viewcomp.asp?comp=8034> jul 2008

AGUIAR, Vicente Macedo - Os Argonautas da Internet: UMA ANÁLISE NETNOGRÁFICA SOBRE A COMUNIDADE ON-LINE DE SOFTWARE LIVRE DO PROJETO GNOME À LUZ DA TEORIA DA DÁDIVA, Salvador, Bahia jul. 2007. Disponível em: <http://www.bibliotecadigital.ufba.br/tde_busca/arquivo.php?codArquivo=727> Acesso em: mai. 2008

BNB Documento de Dispensa de Licitação, Fortaleza, 2003. Disponível em: <http://www.banconordeste.gov.br/content/aplicacao/fornecedores/Editais_Publicados/Editais/DISPENSA_INEX_DEZEMBRO_2003.htm> Acesso em: 23/08/2008

CMMI – CMMi 1.2 Browser Disponível em <http://www.cmmi.de/cmmi_v1.2/browser.html> Acesso em: ago. 2008

COHN, Mike. User stories Applied: for agile software, Addison-Wesley, 2004 <http://agilblog.locaweb.com.br/tag/validacao/> Acesso em: 16/07/2008

IEEE Computer Society, Recommended Practice for Software Requirements Specifcations, 1998. Disponível em: <http://www.csri.utoronto.ca/~sme/CSC2106S/papers/IEEE-STD-830-1998.pdf > Acesso em 24 mai. 2008

ELES, Peter - Capturing Architectural Requirements, IBM, nov. 2005. Disponível em <http://www.ibm.com/developerworks/rational/library/4706.html#footnotes> Acesso em: out. 2007.

FRAMEWORK, 2008. Site no Colab Serpro sobre o Framework de Desenvolvimento Java do SERPRO. Disponível em: <http://colab.serpro/projects/frwjavaintcont/> Acesso em 22 ago. 2008.

GOMES, Wagner – Wagner Gomes's Weblog, 2008. Disponível em: <http://wagnergomes.wordpress.com/category/engenharia-de-software>

GROSSO, Carlos - Verificação e Validação no CMMI, 2006. Disponível em <http://www.sinfic.pt/SinficNewsletter/sinfic/Newsletter86/Dossier3.html> Acesso em: ago. 2008.

HALBLEIB, Harold - Requirements Management, 2004. Disponível em: <http://www.excelsoftware.com/requirements_management.pdf> Acesso em: ago. 2008.

HEUMANN, James – White Paper: Tips for writing good use cases, 2008. Disponível em <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/RAW14023-USEN-00.pdf> Acesso em: jul, 2008.

INCOSE Requirements Management Tools Survey. Disponível em <http://www.paper-review.com/tools/rms/read.php> Acesso em: 20/09/2007

LERDORF, Rasmus – Large Scale PHP FISL08 <http://talks.php.net/show/fisl08> Abr. 2008

MOSKITT, 2008. Modeling Software Kitt. Site sobre a ferramenta MOSKitt. Disponível em <http://www.moskitt.org/>, Acesso em: 18 ago. 2008.

NUSEIBEH, Bashar; EASTERBROOK, Steve – Requirements engineering: a roadmap, 2000. Disponível em <http://www.cs.toronto.edu/~sme/papers/2000/ICSE2000.pdf> Acesso em jul. 2008.

PSDS – Processo Serpro de Desenvolvimento de Soluções. Disponível em: <http://psds.portalcorporativo.serpro> Acesso em: 11 ago. 2008

RAYMOND, Eric Steven. A Catedral e o Bazar Disponível em: <http://www.geocities.com/CollegePark/Union/3590/pt-cathedral-bazaar.html>. Acesso em: 20 de ago.de 2008.

RUP – IBM Rational Unified Process. "Rational Unified Process". Disponível em: <http://www-306.ibm.com/software/awdtools/rup/>. Acesso em: ago. 2008.

SCRUM - Disponível em: <http://www.improveit.com.br/scrum> Acesso em: mai. 2008.

TESTLINK, 2008. TestLink. Disponível em: <http://testlink.org/wordpress/>, Acesso em: 18 ago. 2008.

TRTRIO Especificação Técnica para Aquisição de Ferramentas IBM Rational para o desenvolvimento de aplicações do TRT no RJ, Rio de Janeiro, nov. 2006. Disponível em <http:www.trtrio.gov.br/Administrativo/Licitacoes/PROJETO%20BASICO%20AQ.FERRAMENTAS%20IBM%20RATIONAL.doc> Acesso em: 23 ago. 2008.

WIKIPEDIA, Enciclopédia livre. Disponível em <http://pt.wikipedia.org>

ZANLORENCI, P. Edna; BRUNETT, C. *Robert* - ENGENHARIA DE REQUISITOS Bate Byte, 1998 <http://www.pr.gov.br/batebyte/edicoes/1998/bb77/engenharia.htm>

Padronização do desenvolvimento de aplicações corporativas por meio de um arcabouço de software baseado em uma arquitetura de referência - Demoiselle

Flávio Gomes da Silva Lisboa
Serviço Federal de Processamento de Dados
flavio.lisboa@serpro.gov.br
Palavras-chave: aplicações, arcabouço, arquitetura, Java, padrões.

Introdução

Neste artigo apresentaremos o escopo do Demoiselle Framework, plataforma livre e aberta para desenvolvimento padronizado de aplicações. Inicialmente veremos o escopo do framework e exatamente em que ele consiste. Depos apresentaremos os objetivos e restrições arquiteturais sobre os quais ele foi construído, e em seguida, a perspectiva estrutural da solução adotada. Logo após, uma ênfase sobre a extensão baseada em componentes. Finalmente, teremos uma visão geral de uma aplicação por meio de uma construção que ilustra o uso do plugin Demoiselle Wizard para Eclipse.

Escopo

Demoiselle Framework é uma solução para desenvolvimento de aplicações em Java construída pelo Serpro, para padronizar a construção de suas aplicações e promover a geração de componentes de software reutilizáveis. Ele consiste em duas partes, um framework arquitetural e as dependências estruturais desse framework. O foco deste artigo é o framework arquitetural, mas apresentaremos uma visão geral dessas duas partes, para justificar a escolha da plataforma utilizada.

Um framework pode ser compreendido como "um conjunto de classes cooperativas que implementam os mecanismos que são essenciais para um domínio de problemas específicos" (Horst07, p. 312) e que "constroem um projeto reutilizável para uma determinada categoria de software" (GHJV00, p. 41).

Gamma (GHJV00, p. 332) ainda afirma que "um framework fornece uma melhor orientação arquitetônica do software, através do particionamento do projeto em classes abstratas e da definição de suas responsabilidades e colaborações". Segundo Gamma, o desenvolvedor pode customizar o framework "para uma aplicação particular, através da especialização e da composição de instâncias de suas classes".

Demoiselle Framework, além das definições acima, implementa o conceito de framework integrador (Maci08). Isso ficará mais evidente adiante, quando for tratada a camada dos frameworks especialistas.

A estrutura geral do Demoiselle é baseada na divisão em camadas, pois, segundo Fowler (Fowl06, p. 37-38), isso permite a compreensão de uma parte do sistema como um todo coerente sem a necessidade de saber muito sobre as demais partes. Além disso, a utilização de camadas permite a fácil substituição de implementações, a minimização de dependências entre partes das aplicações, e a definição de padrões que podem gerar código reutilizável.

A camada mais baixa do Demoiselle é o sistema operacional. Essa camada não é passível da imposição de padrão, porque a escolha do sistema operacional depende de vários fatores, como os tipos de aplicação que serão executadas, a necessidade de customização (adaptação do sistema) e a disponibilidade de recursos financeiros (para pagamento de licenças e contrato de suporte) ou humanos (para manutenção do sistema por conta própria).

A imposição de um sistema operacional para um framework voltado ao governo é inviável, pois aprisiona o Estado a uma tecnologia e tende a onerar o cidadão. Dado isso, as aplicações devem ser capazes de rodar em qualquer sistema operacional (ou pelo menos na maioria disponível/usada no mercado). A tecnologia Java permite a portabilidade pela implementação do conceito de máquina virtual, que constitui a segunda camada mais baixa do Demoiselle.

A próxima camada mais acima é denominada plataforma. Como a implementação atual do Demoiselle está direcionada a construção de aplicações Web, a camada de plataforma se constitui de servidores de

aplicação Web que sigam a especificação Servlets 2.5 (JCP00). A criação de aplicações que não sejam Web (Desktop, embarcadas em dispositivos móveis) implicará em mudanças nessa camada. Em seus projetos, o Serpro fez uso de JBoss e Tomcat.

A próxima camada mais superior, dos frameworks de fundamento, constitui-se da reunião de especificações da tecnologia Java que norteiam as implementações de software utilizadas pelo framework. A idéia é que as aplicações não fiquem presas a produtos de software específicos. A adoção dos padrões definidos pelas especificações garante uma certa liberdade na troca de tecnologias especialistas.

As tecnologias especialistas, ou melhor, os frameworks especialistas, fazem parte da camada seguinte. Este é o ponto de mutação do Demoiselle Framework. Os frameworks especialistas são integrados pela última camada no topo, o framework arquitetural, que constitui efetivamente o Demoiselle. O gerenciamento de mudanças deve se concentrar nessas duas camadas superiores, sendo que o framework arquitetural deve permanecer estável.

A idéia é proteger as aplicações do impacto das mudanças, conservando a mesma interface para as classes e métodos utilizados, mas alterando os componentes que implementam as funcionalidades conforme for necessário. O intento é tirar do desenvolvedor a preocupação com as implementações de baixo nível, que constituem a infraestrutura de software.

Todas as tecnologias integradas pelo Demoiselle podem ser perfeitamente usadas sem o framework. Porém, nesse caso, o desenvolvedor divide a sua preocupação com diversas estruturas, precisa gerenciar isso por conta própria e torna as aplicações dependentes de determinadas tecnologias, fortemente sujeitas à mudança. Ao programar para o Demoiselle Framework, o desenvolvedor consegue criar uma estrutura de software reutilizável, de duas formas. Primeiro, se algum framework especialista tiver de ser substituído, a aplicação irá ignorar essa mudança, pois sua interface com o framework permanecerá inalterada. A mudança ocorrerá na implementação do framework que faz a integração, sendo que o resultado final permanecerá o mesmo.

Objetivos e Restrições Arquiteturais

Doravante, quando nos referirmos a Demoiselle, estaremos tratando do framework arquitetural. A arquitetura do Demoiselle é norteada e restringida

por um conjunto de objetivos, a saber, extensibilidade, reusabilidade, manutenibilidade, desempenho, estabilidade e confiabilidade.

A arquitetura atual foi desenhada para atender somente aplicações Web não distribuídas.

Extensibilidade

Seria absurdo presumir a possibilidade da criação de uma infraestrutura de software que pudesse atender às demandas de todas as aplicações. Essa é uma dificuldade do estabelecimento e aceitação de padrões. Tentativas de cercar todos os problemas tendem ao fracasso, pois não é possível prever as funcionalidades necessárias para atender aplicações futuras.

De modo a convergir a padronização das aplicações, pré-requisito para o objetivo tratado a seguir, e a capacidade dos sistemas de crescerem, a arquitetura do framework define a existência de pontos de extensão seja por meio de interfaces, abstrações ou pela utilização de padrões de projetos.

As principais interfaces da arquitetura são:

IDAO: Interface para as classes que implementam o padrão de projetos DAO (Alcm02, p. 346), no qual um objeto é responsável por extrair e encapsular todos os acessos à origem de dados e gerenciar a conexão com a origem de dados para obter e armazenar dados. Classes que implementam IDAO não acessam outras camadas da aplicação, apenas a de persistência;

IFacade: Interface para classes que implementam o padrão Façade (GHJV00, p. 179), cujo objetivo é unificar subsistemas por meio de uma interface de nível mais alto e tornar mais fácil o uso dos mesmos;

IBusinessController: Interface para classes que definem objetos da camada de negócios (AlCM02, p. 50) que acessam a camada de persistência através de classes que implementam a interface IDAO. Objetos que implementam IBusinessController podem acessar funcionalidades de outros sistemas ou módulos através de implementações de IFacade;

IViewController: Interface para classes que fundem os conceitos de visão e controle do MVC (Fowl06, p. 315). Objetos dessas classes terão

acesso somente a instâncias de classes que implementam IFacade e IBusinessController;

Uma característica inerente dos frameworks, citada por Fowler (Fowl05) é que "os métodos definidos por um usuário para adaptar o framework às suas necessidades são frequentemente chamados de dentro do próprio framework, em vez do código de aplicação do usuário. O framework frequentemente faz o papel do programa principal na coordenação e sequenciamento de atividades da aplicação. Essa inversão de controle dá aos frameworks o poder de servir como esqueletos extensíveis. Os métodos fornecidos pela adaptação do usuário para os algoritmos genéricos definidos no framework para uma aplicação particular".

Reusabilidade

Segundo Paula Filho (Paul09, p. 256), "quando uma organização começa a usar processos definidos de desenvolvimento de software, os maiores ganhos iniciais resultam da redução dos defeitos introduzidos em cada iteração. Isso ocorre por causa da redução do desperdício de tempo e dinheiro, principalmente aquele que é causada por defeitos de requisitos, análise e desenho. A partir daí, ganhos significativos de produtividade só são conseguidos por meio da reutilização".

Em face disso, a arquitetura de uma infraestrutura de software deve favorecer o reuso, a partir da especificação de artefatos comuns em diversos projetos. O Demoiselle Framework propõe dois principais artefatos de reuso, uma arquitetura de referência e componentes fracamente acoplados.

A arquitetura de referência é uma estrutura padrão de aplicação em camadas. A partir do padrão de projeto MVC (Fowl06, p. 315) e do modelo J2EE (AlCM02, p. 114), o Demoiselle propõe três camadas: Visão e Controle, Negócios e Persistência. Visão e Controle é a fusão das camadas homônimas do padrão MVC, em um entendimento de que são indissociáveis, exceto em aplicações servidoras de serviços. Negócios centraliza a maior parte do processamento de negócios para a aplicação. Persistência corresponde a camada de Modelo do MVC e de Integração do J2EE.

Camadas podem ser reaproveitadas entre aplicações, geralmente com adaptações. O Demoiselle Framework propõe a redução ao mínimo de adaptações com o uso de injeção de dependências (Fowl04). Esse é um padrão cuja idéia básica consiste em um objeto separado, um montador, que

realiza a instanciação da implementação apropriada de uma interface, de modo que uma camada não dependa de classes específicas. A injeção de dependências no Demoiselle é feito com o uso de orientação a aspectos, implementada por AspectJ.

A extensibilidade por meio de componentes fracamente acoplados serve também a reutilização. A proposta do Demoiselle é que as aplicações sejam criadas com blocos de código pré-fabricados, que encerrem funcionalidades completas. O desenvolvimento orientado a componentes favorece o framework quando preserva sua extensibilidade, ao mesmo tempo em que acelera o desenvolvimento e aumenta a qualidade do código, ao induzir a construção de aplicações pela composição de funcionalidades.

A criação de componentes para o Demoiselle segue a orientação de McConnell (Mcco05, p. 78), na qual "a responsabilidade de cada bloco de construção deve ser bem-definida. Um bloco de construção deve ter uma área de responsabilidade e deve saber o mínimo possível sobre as áreas de responsabilidade de outros blocos de construção (fraco acoplamento)".

Manutenibilidade

A arquitetura divide responsabilidades entre módulos lógicos, que serão vistos adiante, com o intuito de causar o menor impacto no todo em caso de manutenção das aplicações. Isso é alcançado com duas abordagens: diminuição do acoplamento e foco da manutenção em pontos específicos.

Desempenho

O projeto do Demoiselle identificou, com base na experiência das equipes de desenvolvimento do Serpro, pontos críticos de performance, tais como integração entre camadas e controle de transações. De modo a diminuir os riscos de perda de desempenho, na manutenção e evolução de aplicações, o framework implementa os pontos críticos como funcionalidades, que fazem parte dos pontos específicos de manutenção citados anteriormente.

Estabilidade e Confiabilidade

Estabilidade aqui não se refere à garantir a execução das aplicações, mas sim que seu comportamento continue sendo o esperado após a realização

de manutenções. Para garantir isso, foi decidido que o framework não dependeria de implementações específicas de determinadas funcionalidades, e sim que ele seria fundamentado em especificações tecnológicas. Desse modo, as aplicações ficam protegidas de mudanças em produtos de software, pois as implementações podem ser substituídas por outras que sigam as mesmas especificações.

A estabilidade do framework garante a confiabilidade das aplicações, pois ao manter as mesmas condições estabelecidas (especificações), a tendência é que o sistema prossiga normalmente na execução de suas funções.

Perspectiva Estrutural da Solução

Apresentaremos aqui uma visão lógica do framework arquitetural, os módulos lógicos nos quais está dividido e quais os elementos de projeto presentes nesses módulos. O framework consiste em cinco módulos: core, util, web, persistence e view. Esses módulos são bibliotecas de classes Java, disponibilizadas independentemente como arquivos .jar.

Módulo Core

Este é o módulo que contém o conjunto de especificações que dão base estrutural ao framework. Seu objetivo é tornar possível a padronização, extensão e integração entre as camadas das aplicações baseadas no framework.

O pacote br.gov.framework.demoiselle.core.layer define as abstrações para os objetos de cada camada. Ele contém as interfaces vistas nos objetivos e restrições arquiteturais: IViewController, IBusinessController, IDAO e IFacade. À exceção de IDAO, as demais interfaces não contém declaração de métodos, o que pode parecer estranho, já que interfaces servem para definir comportamentos padronizados por meio de código genérico reusável.

O que ocorre é as interfaces do módulo Core possuem outro propósito: servir para identificar os pontos de injeção de dependências.

Geralmente as aplicações farão uso das três primeiras interfaces. IFacade serve para definir uma fachada quando a aplicação trata de integração entre módulos ou integração de subsistemas.

O Core não possui nenhuma funcionalidade imediatamente utilizável. Ele apenas define os padrões do framework, de modo que outros módulos tem de estendê-lo para tornar o framework funcional.

Integração entre Camadas

O pacote br.gov.framework.demoiselle.core.layer.integration define uma abstração para o mecanismo de integração entre camadas.

O mecanismo de integração atua na camada de visão injetando objetos de negócio por meio de uma fábrica (GHJV02, p. 95) do próprio framework ou alguma fábrica definida pela aplicação. Essa fábrica pode utilizar um proxy (GHJV02, p. 198) do framework ou da aplicação para instanciar objetos de negócio.

De forma similar, na camada de regras de negócio, o mecanismo atua injetando objetos de persistência.

Três interfaces são definidas neste pacote: IFactory, IBusinessControllerFactory e IDAOFactory.

IFactory é uma abstração de fábricas de objetos injetados. IBusinessControllerFactory e IDAOFactory são especializações de IFactory, respectivamente para objetos de negócio e objetos de persistência.

Beneficiando-se das funcionalidades oferecidas pela versão 5 de Java, o framework define no módulo Core duas anotações (annotations): Injection e Factory. A primeira é uma anotação de propriedade que contém informações a respeito da injeção de dependência. A segunda é uma anotação de classe ou pacote usada para definir a implementação da fábrica utilizada na injeção de dependência.

A classe InjectionContext responsabiliza-se por manter as informações necessárias para a fábrica realizar a criação de objetos.

O módulo Core especifica quem trata a injeção de dependência, mas a forma como a injeção será realizada deve ser definida pelos módulos que implementam as abstrações do Core. Na versão 1.0.x do Demoiselle, é o módulo Web que implementa a injeção de dependência.

Isso não impede que ilustremos como se dá o fraco acoplamento proporcionado pelo uso desse padrão. O código a seguir é uma classe que implementa a interface IViewController, que não contém nenhum comportamento definido. Isso tão somente funciona como um marcador, ou na linguagem da AOP (Aspect Oriented Programming), um join point. Isso serve para avisar ao compilador da linguagem orientada a aspectos, que é executado antes do compilador Java ordinário, que um aspecto pode ser aplicado neste ponto, ou mais especificamente, nessa classe.

Observe que a classe MeuMB define um atributo cujo tipo não é uma classe, e sim uma interface. IMeuBC é uma extensão de IBusinessController e também constitui um join point. Antes do atributo, porém, há uma anotação @Injection. O que vai ocorrer quando da compilação dessa classe, é que o compilador de aspectos identificará pela anotação que o atributo precisa receber uma instância. Aí ocorre a injeção de código. Na linha imediatamente a seguir ao da declaração é incluído um pedaço de código (advice) que faz a atribuição de uma classe que implementa IMeuBC para meuBC.

```
public class MeuMB implements IViewController{
        @Injection
        private IMeuBC meuBC;
}
```

A classe que será instanciada é definida pelo módulo que implementa o Core. Pode-se induzir que a classe tenha que seguir um padrão de nome para ser localizada. E isso pode levar a crer que é difícil flexibilizar a aplicação para que determinados atributos instanciem classes fora do padrão. Mas é facil criar exceções à regra, pela passagem do parâmetro name para a anotação @Injection, como se vê na classe MeuMB modifica a seguir.

```
public class MeuMB implements IViewController{
        @Injection
 (name=”br.gov.escola.business.implementation.AlunoBC”)
        private IMeuBC meuBC;
}
eee  o/c
```

Contexto de Mensagens

O pacote br.gov.framework.demoiselle.core.message define uma abstração de mensagens trocadas durante uma requisição entre as camadas dos sistemas. Essa abstração permite criar um contexto de mensagens para que todas as camadas definidas pelas arquitetura de referência possam manipular mensagens. Ela também auxilia a exibição das mensagens para o usuário, seja na forma de tela, arquivo texto (log), em banco de dados ou na junção dessas opções. Cada implementação

dessa especificação deve prover uma solução de acesso ao contexto de mensagem.

Esse pacote define duas interfaces, IMessage e IMessageContext. A primeira é uma abstração da unidade de mensagem, enquanto a segunda abstrai o contexto de mensagem. O pacote ainda conta com um tipo enumerado chamado Severity, que padroniza as severidades genéricas.

Exceção Padronizada

O pacote br.gov.framework.demoiselle.core.exception define a classe ApplicationRuntimeException como a exceção padrão para ser utilizada pelas aplicações. Essa classe encapsula uma instância de IMessage, constituindo uma mensagem de erro padronizada que pode ser tratada facilmente pelos módulos do framework. ApplicationRuntimeException é uma exceção não verificada, portanto não precisa obrigatoriamente ser capturada ou declarada.

A seguir temos um exemplo de lançamento da exceção padronizada, com a passagem de uma mensagem definida pelo usuário.

```
public void MetodoBC(){
      if ( /*Condição para lançamento de exceção*/ ){
            throw new
ApplicationRuntimeException(ErrorMessage.ERRO_01);
      }
}
```

Contexto de Segurança

O framework especifica uma forma padronizada de acesso aos dados da aplicação com restrições de segurança por intermédio de autenticação e autorizada baseada em papéis. Cada implementação desta especificação do Core deve prover uma solução de acesso ao contexto de segurança. A abstração do contexto de segurança é baseada na especificação JAAS, uma API que permite que aplicações escritas na plataforma J2EE usem serviços de controle de autenticação e autorização sem necessidade de estarem fortemente dependentes desses serviços.

A seguir temos um exemplo de uso do contexto de segurança:

```
ISecurityContext contexto =
ContextLocator.getInstance().getSecurityContext();
if (contexto.isUserInRole("Administrador")){
...
}
```

A utilização do método getInstance() assinala o uso do padrão de projeto Singleton (GHJV00, p. 130), o que é coerente, visto que as informações relativas a segurança devem ficar centralizadas. A classe que implementa esse Singleton será tratada adiante.

Entidades

O pacote br.gov.framework.demoiselle.core.bean propõe uma abstração para as entidades da aplicação, a interface IPojo, que força a utilização do padrão Value Object (AlCM02, p. 232).

Contexto de Transação

O pacote br.gov.framework.demoiselle.core.transaction especifica o mecanismo de controle transacional e define um contexto de transação que atua no início e no fim de cada ação. Seu funcionamento depende de um tipo definido pela enumeração TransactionType, que pode ser LOCAL ou JTA (Java Transaction API). LOCAL indica que a aplicação será responsável pelo gerenciamento da transação, enquanto JTA indica que a aplicação dependerá de uma implementação JTA disponível no contêiner.

Esse pacote possui duas interfaces, ITransactionResource e ITransactionContext. A primeira define um recurso a ser registrado no contexto de transação, enquanto a segunda representa o contexto de transação responsável por registrar o início e o fim de cada ação e registrar os recursos transacionais.

Acionadores

O pacote br.gov.framework.demoiselle.core.action define um mecanismo padronizado de ações a serem executadas pela aplicação. Essas ações são definidas como funções estruturas da aplicação, como carregamento de configuração e inicialização de ambiente.

As interfaces desse pacote são IActionManager, ILoaderAction e IAction. A primeira define um padrão para classes que gerenciam ações, com métodos para configurar o carregador da ação (que deve implementer ILoaderAction). O carregador da ação contém uma coleção de tipos IAction, que per sua vez representam ações.

Um exemplo de ação padronizada pode ser visto a seguir:

```
public class MinhaAplicacaoAction implements IAction {
private static Logger log =
 Logger.getLogger(MinhaAplicacaoAction.class);
        public void execute() {
                log.debug("Lendo arquivos de configuração");
        }
}
```

Uma classe que implemente ILoaderAction, por exemplo MeuCarregadorAction, recupera as ações com seu método getActions(), que retorna uma coleção do tipo IAction. Uma implementação de IActionManager, por sua vez, dispara todas as ações pela chamada a seus métodos execute(), algo extremamente simples para a estrutura for trazida pelo Java 5, que itera sobre coleções.

Localizador de Contextos

O pacote br.gov.framework.demoiselle.core.context define uma classe chamada ContextLocator. Essa classe é fundamental para que as aplicações consigam ter um acesso fácil e centralizado aos contextos de Mensagem, Transação e Segurança, e ao mesmo tempo garante que eles tenham, cada um, uma única instância, pela aplicação do padrão Singleton.

As implementações de cada contexto devem obrigatoriamente utilizar o localizador de contextos como canal de acesso, para que todas as camadas da aplicação possam utilizar os contextos.

Módulo Util

Este módulo que contém componentes utilitários da maior amplitude genérica, que visam facilitar o trabalho de outras funcionalidades do framework.

Na versão 1.0.x consiste de dois pacotes, um para carregamento de configuração e outro para paginação de dados.

Carregamento de Configuração

O pacote br.gov.framework.demoiselle.util.config define um mecanismo padronizado de carregamento de configuração que permite carregar variáveis configuradas no ambiente, arquivos XML ou arquivos de propriedades para um objeto. Esse mecanismo é utilizado em vários outros componentes do framework e pode ser utilizado também pelas aplicações.

A classe ConfigurationLoader é o utilitário que executa o carregamento das configurações nas classes. Uma anotação chamada ConfigKey é usada nas classes para identificar os atributos que podem ser carregados a partir de uma configuração. A limitação dos recursos aceitos pelo mecanismo é feita pelo tipo enumerado ConfigType. Como sério candidato a exceções, o pacote contém uma exceção própria, ConfigurationException, que herda de RuntimeException, o que a torna também não verificável.

O trecho de código a seguir traz exemplos das declarações de carregamento de configuração possíveis:

```
@ConfigKey (name = “key”, type=ConfigType.SYSTEM)
private String stringValueSystem;

@ConfigKey (name = “framework.stringValue”,
 type=ConfigType.XML,
resourceName=”configuration.xml”)
private String stringValueXML;

@ConfigKey (name = “framework.stringValue”,
 type=ConfigType.PROPERTIES,
resourceName=”configuration.properties”)
private String stringValueProperties;
```

Esses atributos poderiam, por exemplo, pertencer a uma classe chamada MeuConfig. O carregamento e uso da configuração se daria da forma a seguir:

```
public void meuMetodo() {
        MeuConfig meuConfig = new MeuConfig();
        ConfigurationLoader.load(meuConfig);
        System.out.print( meuConfig.getMinhaPropriedade());
}
```

Paginação de Dados

As aplicações geralmente necessitam trafegar resultados entre as camadas de forma paginada garantindo o desempenho da aplicação. Esse mecanismo é implementado no pacote br.gov.framework.demoiselle.util.page com um objeto que permite configurar os dados da página que será requisitada (instância de Page) e um objeto que contém os resultados de forma paginada (instância de PageResult).

Módulo Web

Este módulo contém implementações de algumas especificações do módulo Core e também utilitários comuns de aplicações Web não distribuídas que facilitam o tratamento de sessões de usuário e suas requisições.

Contexto de Segurança

O pacote br.gov.framework.demoiselle.web.security define duas classes para implementar o mecanismo de acesso aos dados com segurança, WebSecurityContext e WebSecurityServletRequestListener. A primeira implementa o contexto de segurança através do padrão Singleton, além de gerenciar os dados de segurança vinculados a thread corrente. A segunda é responsável por repassar o objeto “request” para a instância da primeira.

Contexto de Mensagens

O pacote br.gov.framework.demoiselle.web.security define a classe WebMessageContext para implementar o contexto de mensagens. O código a seguir ilustra o lançamento de mensagens para o contexto:

```
IMessageContext contextoMsg =
ContextLocator.getInstance().getMessageContext();
public class MeuBC implements IBusinessController {
      public void meuMetodo(){
            contextoMsg.addMessage(InfoMessage.Mensagem);
      }
}
```

O método getMessageContext() devolve uma instância de WebContextMessage, mas isso não importa para a aplicação, desde que a classe implemente IMessageContext. Isso permite que esse código fique protegido contra mudanças caso o contexto de mensagens seja modificado.

A recuperação de mensagens pode ser feita pela chamada ao método getMessages() do contexto de mensagens, que retorna uma coleção de instâncias de IMessage.

Integração entre Camadas

O módulo Web implementa a especificação de integração de camadas proposto pelo módulo Core no pacote br.gov.framework.demoiselle.web.layer.integration. O mecanismo implementado utiliza AOP para detectar os pontos de integração. Permite que sejam implementadas novas fábricas de acordo com a necessidade da aplicação. Os objetos são fabricados por meio de convenção de nomes entre as interfaces e sua implementação.

WebAbstractFactory é a abstração de uma fábrica genérica do módulo Web, seguindo o padrão de projeto Abstract Factory (GHJV00, p. 95). Todas as outras fábricas desse módulo devem especializar essa classe. WebBusinessControllerFactory e WebDAOFactory implementam respectivamente as interfaces IBusinessController e IDAOFactory, definidas no Core.

A classe responsável pelo mecanismo de injeção de dependências é InjectionManager que faz uso das definições de injeção do Core. InjectionManager é chamada quando o aspecto InjectionAspect intercepta instanciações de classes que implementam IBusinessController, IViewController e Ifacade.

O subpacote br.gov.framework.demoiselle.web.layer.integration oferece uma implementação opcional para desenvolver objetos criados pela injeção

com um proxy (GHJV00, p. 198). A interface IProxy define o comportamento padrão do proxy chamado pela injeção de dependência (InjectionManager). Sua implementação deve ser registrada no arquivo de configuração do framework e carregada através da classe ProxyConfig.

A classe WebProxy é a implementação básica da interface IProxy. Uma classe de configuração chamada ProxyConfig diz qual classe implementa a interface IProxy. A injeção de dependências faz uso do controlador de chamadas de métodos WebInvocationHandler na hora de envolver o objeto criado em um proxy.

Contexto de Transação

O módulo Web implementa a especificação do contexto transacional proposto no módulo Core. O mecanismo implementado utiliza AOP para prover um mecanismo transparente de gerenciamento de transação.

É possível utilizar o controle transacional do contêiner (JTA). Para isso deve existir uma implementação de um mecanismo de lookup via JNDI. A interface IJNDITransactionManagerLookup define as informações para o mecanismo JNDI localizar uma UserTransaction do contêiner com suporte JTA. O módulo Web provê uma implementação para essa interface, voltada para o JBoss: JbossTransactionManagerLookup.

A implementação padrão do contexto transacional é a classe WebTransactionContext. A classe WebTransactionAction define a ação de inicialização em aplicações Web onde o contexto transacional é configurado. WebTransactionActionConfig define as configurações padrão do contexto transacional definido pela aplicação em arquivo externo.

A classe WebTransactionServletRequestListener é o controlador do contexto transacional. Ela é responsável por iniciar e finalizar, normalmente ou com erro, o contexto transacional. É acionado em todas as requisições Web.

Inicialização do Ambiente

A inicialização de ambiente implementa a especificação de ações proposta no módulo Core, essa inicialização ocorre sempre que o contêiner iniciar a aplicação. O módulo Web necessita que algumas ações sejam executadas e essas ações estão implementadas no pacote br.gov.framework.demoiselle.

web.init. Os componentes e aplicações baseadas no framework podem implementar outras ações e adicioná-las para que sejam executadas na inicialização do ambiente.

A interface IinicializationAction define o comportamento de uma ação de inicialização de aplicações Web. A classe WebInicializationServletContext Listener executa todas as ações configuradas ao inicializar o contêiner web/servlet. A classe WebInicializationLoaderAction carrega todas as classes de ações, baseada nas configurações representadas pela classe WebInicializationLoaderActionConfig.

O código a seguir é um exemplo de classe de inicialização do ambiente:

```
public class MinhaAction implements IInitializationAction {
      public void execute() {
            log.debug("Inicializando minha action");
      }
      public void setServletContext(ServletContext context) {
      }
}
```

O método execute() dessa classe será invocado automaticamente na inicialização, desde que a classe seja referenciada no arquivo demoiselle.properties, conforme exemplo a seguir:

```
framework.demoiselle.web.initialization.action=MinhaAction
```

Redirecionamento Baseado em URL

O módulo Web implementa a especificação do contexto transacional proposto no módulo Core. O mecanismo implementado utiliza AOP para prover um mecanismo transparente de gerenciamento de transação.

A implementação do redirecionamento consiste em cinco passos, que ilustraremos a seguir.

Inicialmente, temos de criar as ações de redirecionamento. Um exemplo é a classe adiante, MinhaRedirectAction:

```
public class MinhaRedirectAction implements IRedirectAction {
      private ServletReq uest request;
```

```
private ServletResponse response;

public String getParameter() {
        return "MinhaActionParameter";
}

public String getValue() {
        return "MinhaActionValue";
}

public void setRequest(ServletRequest req) { this.request = req; }

public void setResponse(ServletResponse resp) { this.response = resp;
}

public void execute() {
/*Minha execução*/
}
}
```

Em seguida, é necessário cadastrar as ações no arquivo demoiselle.properties:

```
# — Web configuration —
framework.demoiselle.web.redirect.action=MinhaRedirectAction01
framework.demoiselle.web.redirect.action=MinhaRedirectAction02
framework.demoiselle.web.redirect.action=MinhaRedirectAction03
```

Posteriormente, a classe WebRedirectServlet deve ser mapeada no arquivo web.xml:

```
<servlet>
        <servlet-name>
        WebRedirectServlet
        </servlet-name>
        <servlet-class>
```

```
br.gov.framework.demoiselle.web.redirect.WebRedirectServlet
        </servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>
        WebRedirectServlet
        </servlet-name>
        <url-pattern>
        /redirect
        </url-pattern>
</servlet-mapping>
```

Finalmente, na página JSP, ou JSF, a ação de redirecionamento é chamada desta forma:

```
<ahref=”minhaAplicacao/redirect?
MinhaActionParameter=MinhaActionValue”>
Chamar Minha Action
</a>
```

Módulo Persistence

O módulo Persistence trata as funções de persistência das aplicações e sua integração com outros módulos. Ele provê a integração da aplicação sistemas gerenciadores de dados e garante transparência às demais camadas da aplicação na recuperação, armazenamento e tratamento das informações.

A interface IDAO do módulo Core, em oposição as outras interfaces do pacote br.gov.framework.demoiselle.core.layer, define quatro operações. Ela é a base para o módulo de persistência do framework.

A persistência no Demoiselle Framework enfoca duas abordagens: o acesso direto por JDBC e o mapeamento objeto-relacional. O acesso a sistemas de bancos de dados não-relacionais, geralmente legado, pode ser feito por componentes.

Um pacote com três classes trata do desenvolvimento com drivers JDBC. JDBCGenericDAO implementa alguns métodos de IDAO, para diminuir o trabalho do desenvolvedor e padronizar o código. A classe JDBCUtil é um utilitário para as configurações JDBC e também a responsável por inserir uma conexão no controle transacional definido pelo módulo Core. Finalmente, a classe JDBCTransactionResource representa uma conexão JDBC e pode ser tratada pelo contexto de transação do framework.

O uso de JDBCUtil para gerenciar conexões pode ser visto neste exemplo de consulta:

```
public class FuncionarioDAO extends JDBCGenericDAO
<Funcionario> implements IFuncionarioDAO {
    public List<Funcionario> listar() {
        List<Funcionario> result = new ArrayList<Funcionario>();
        PreparedStatement prepStmt = null;
        Connection con = null;
        try {
            con = JDBCUtil.getInstance().getConnection();
            prepStmt = con.prepareStatement("SELECT *
FROM Funcionario");
            ResultSet rs = prepStmt.executeQuery();
            while (rs.next()) {
                long id = rs.getLong("id_funcionario");
                String nome = rs.getString("nome");
                Date nascimento = rs.getDate("nascimento");
                String lotacao = rs.getString("lotacao");
                result.add(new Funcionario(id, nome,
nascimento, lotacao));
            }
        } catch (SQLException e) {
            throw new
ApplicationRuntimeException(ErrorMessage.FUNCIONARIO_005,
e);
        } finally {
            JDBCUtil.getInstance().closeConnection();
        }
        return result;
```

```
        }
}
```

Para evitar que a troca de banco implique em alteração no código-fonte das classes, a configuração das conexões JDBC é feita no arquivo demoiselle.properties, como mostra o exemplo a seguir:

```
#Configuração para uso de JDBC
framework.demoiselle.persistence.jdbc.driver=org.postgresql.Driver
framework.demoiselle.persistence.jdbc.url=jdbc:postgresql://localhost/
escola
framework.demoiselle.persistence.jdbc.user=postgres
framework.demoiselle.persistence.jdbc.pass=postgres
```

Para tratar o mapeamento objeto-relacional, o módulo Persistence define uma interface chamada IORMDAO, que estende IDAO e define dois métodos, findByExample() e findById(). É possível integrar qualquer framework especialista de persistência que use ORM, desde que seja definida uma classe que implemente IORMDAO e outra que seja um recurso transacional.

A versão 1.0.x do Demoiselle utiliza o Hibernante como framework de persistência. O pacote br.gov.framework.demoiselle.persistence.hibernate define a integração dos frameworks com três classes: HibernateGenericDAO, HibernateUtil e HibernateTransactionResource.

A matéria-prima da camada de persistência são classes Pojo, que representam as entidades da aplicação. Um exemplo é a classe Aluno, a seguir:

```
package br.gov.framework.demoiselle.escola.bean;
public class Aluno implements IPojo{
        private Long id;
        public Aluno() {}
        public Long getId() {
                return id;
        }
        public void setId(Long id) {
                this.id = id;
```

```
    }
}
```

A arquitetura de referência define para as aplicações um pacote br.gov.demoiselle.nomedaaplicacao.persistence, no qual ficam as interfaces que estendem IDAO e as implementações das mesmas. A classe WebDAOFactory é usada por padrão para construir os objetos por meio de convenção. A convenção é que as interfaces e suas implementações devem ter a mesma identificação, sendo que as interfaces devem ter o prefixo I e o sufixo DAO, enquanto as implementações devem omitir o prefixo. Por exemplo, a interface IAlunoDAO deve ser implementada pela classe AlunoDAO.

É possível modificar a fábrica que fará a injeção de dependências, estendendo WebDAOFactory e usando o parâmetro factory da anotação @Factory, conforme exemplo a seguir:

```
public class EscolaDAOFactory extends WebDAOFactory {
    public IDAO create(InjectionContext ctx) {
        return //Lógica da Fábrica;
    }
}

@Factory(factory=EscolaDAOFactory.class)
public class AlunoDAOStubTest implements IFacade{
    @Injection
    private IAlunoDAO alunoDAO;
}
```

As operações de persistência fazem uso do contexto de transação, conforme vemos no código seguinte:

```
public class MinhaClasse {
    @Injection
    IMeuDAO meuDao;

    public void listar() {
        WebTransactionContext.getInstance().init();
        meuDao.listar();
        WebTransactionContext.getInstance().end();
```

```
        }
}
```

A classe HibernateUtil provê um conjunto de implementações para camada de persistência baseado no framework Hibernate. Ela é integrada ao contexto de transação, de modo que possui um mecanismo transacional completo. O código a seguir ilustra dessa classe para tratar tanto o commit quando o rollback de uma transação:

```
public class MinhaClasse {
        @Injection
        IMeuDAO meuDao;

        public void inserir() {
                WebTransactionContext.getInstance().init();
                try{
                        meuDao.insert(new MeuPojo());
                        HibernateUtil.getInstance().commit();
                }catch (ApplicationRuntimeException e) {
                        HibernateUtil.getInstance().rollback();
                }
                WebTransactionContext.getInstance().end();
        }
}
```

A paginação de dados definida pelo módulo Util é usada pelo módulo Persistence para manter o controle sobre os dados oriundos de uma consulta e recuperar da base apenas as informações que serão exibidas na visão do usuário. Os benefícios da paginação são a redução de custos de uso de memória, processamento e rede e a escalabilidade proporcionada. O código a seguir ilustra o uso de paginação. O trecho de código é de uma classe herdeira de HibernateGenericDAO:

```
public void listar() {
        Page page = new Page(50, 1);
        listarBean(page);
}
```

```
public PagedResult<Aluno> listarBean(Page page) {
        return findHQL("from Bean", page);
}
```

HibernateGenericDAO implementa um conjunto de funções (consulta, paginação, inserção, alteração e exclusão) para simplificar a criação de classes IDAO implementadas pela aplicação. Um exemplo de como se fazer consultas com HibernateGenericDAO pode ser visto no código a seguir:

```
public class DisciplinaDAO extends HibernateGenericDAO
<Disciplina>
implements IdisciplinaDAO{

        public PagedResult<Disciplina> listar(Page page) {
                return findHQL("from Disciplina order by nome asc",
page);
        }

        public Disciplina buscar(Disciplina professor) {
                List<Disciplina> retorno = find("from Disciplina order by
nome asc");
                if (retorno != null && retorno.size() > 0 )
                        return retorno.get(0);
                return null;
        }

        public PagedResult<Disciplina> filtrar(Disciplina prof, Page
page) {
                return findByExample(prof, page);
        }

        public List<Disciplina> listar() {
                return findHQL("from Disciplina order by nome asc");
        }
}
```

Módulo View

Este módulo contém implementações de componentes específicos de interface com usuário baseados na especificação JSF. Uma vez que a especificação JSF define a utilização de um controlador MVC denominado managed bean, o módulo View disponibiliza uma implementação padronizada para ser utilizada nas aplicações. É a classe AbstractManagedBean.

A classe AbstractManagedBeanConfig representa as configurações da aplicação do total de linhas e a quantidade de páginas ao paginar dados.

Para facilitar as operações com managed beans, o framework disponibiliza a classe ManagedBeanUtil. O módulo View também provê a classe PagedResultDataModel, um modelo de dados que converte um PagedResult para a representação gráfica dos dados paginados.

Finalmente, o módulo View define a classe CookieManager, um utilitário que permite a realização das operações básicas relacionadas a cookies na Web.

Extensão por Componentes

O framework, enquanto distribuição de software, está dividido em dois módulos principais, o primeiro é o framework arquitetural, que foi tratado até agora, e o segundo é um conjunto de componentes. O framework arquitetural promove a padronização na construção das aplicações. Os componentes são complementares ao framework e possuem ciclo de vida próprio, desta forma podem ser utilizados individualmente de acordo com a necessidade da aplicação. Novos componentes podem ser adicionados a cada release.

O Serpro disponibilizou uma série de componentes como software livre dentro de um projeto vinculado ao Demoiselle Framework, o Demoiselle Components. Esses componentes podem ser modificados para dar origem a outros, assim como novos componentes podem ser acrescentados ao projeto, estendendo as funcionalidades do framework.

Deve-se evitar ao máximo incluir novas funcionalidades diretamente no framework arquitetural, pois isso tende a diminuir o caráter genérico de sua implementação. Um exemplo é o componente Demoiselle Hibernate Filter. O Serpro utiliza Hibernate em suas aplicações, e inclusive este é o único framework ORM (Object Relationship Mapping) integrado na versão 1.0.x. Mas criar funcionalidades baseadas exclusivamente no Hibernate tornaria o

framework dependente deste, e esse não é o objetivo. Assim, para suprir a necessidade do uso de uma interface de filtros de pesquisa mais elaborada, foi criado o componente Demoiselle Hibernate Filter.

Esse componente permite a definição de conjuntos de campos a serem filtrados, ordenação ascendente e descendente e definição de conjuntos de critérios de pesquisa. Um exemplo de filtro de pesquisa e seu uso pode ser visto no trecho de código seguinte:

```
package br.gov.framework.demoiselle.escola.persistence.dao.filter;
import br.gov.framework.component.demoiselle.hibernate.filter.Filter;
import br.gov.framework.demoiselle.escola.bean.Aluno;
public class FiltroAluno extends Filter {
        private static final long serialVersionUID = 1L;
        public static final String ID = "id";
        public static final String NOME = "nome";
        public static final String USUARIO = "usuario";

        public FiltroAluno(){
                setClazz(Aluno.class);
                addOrder(NOME, ASC);
        }
}

public class DisciplinaDAO extends
HibernateGenericDAO<Disciplina>
implements IDisciplinaDAO{
        public Aluno buscarAluno(Aluno arg0) {
                FiltroAluno filtro = new FiltroAluno();
                filtro.addEquals(FiltroAluno.ID, arg0.getId());
                List<Aluno> retorno = find(filtro);
                if (retorno != null && retorno.size() > 0 )
                        return retorno.get(0);
                return null;
        }
}
```

Plugin para a Plataforma Eclipse

O ambiente de desenvolvimento integrado padrão do Serpro para Java é o Eclipse. Por isso, foi criado um plugin para essa plataforma que permisse a geração de código com os padrões do Demoselle. Isso deu origem ao projeto Demoiselle Wizard. Apresentaremos aqui a construção de uma aplicação com o uso desse Wizard, tanto para demonstrar seu uso como para dar uma visão geral sobre uma aplicação com a arquitetura de referência do Demoiselle.

Nossa base de dados

Utilizaremos para este exemplo um banco de dados criado em PostgreSQL. A listagem seguinte mostra o script para criação do banco rh no PostgreSQL e das tabelas departamento e funcionário.

```
CREATE DATABASE rh
  WITH OWNER = postgres
     ENCODING = 'UTF8';

CREATE TABLE departamento
(
  id serial NOT NULL,
  nome text NOT NULL,
  CONSTRAINT departamento_pkey PRIMARY KEY (id)
)
WITH (OIDS=FALSE);
ALTER TABLE departamento OWNER TO postgres;

CREATE TABLE funcionario
(
  id serial NOT NULL,
 nome text NOT NULL,
  id_departamento integer NOT NULL,
  CONSTRAINT funcionario_pkey PRIMARY KEY (id),
  CONSTRAINT funcionario_id_departamento_fkey FOREIGN KEY
(id_departamento)
```

```
    REFERENCES departamento (id) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (OIDS=FALSE);
ALTER TABLE funcionario OWNER TO postgres;
```

Criação do Projeto

No Eclipse, entramos no menu File->New->Project, e selecionamos o tipo de projeto Demoiselle Project. O nome do projeto será rh.

Criação dos Beans

Agora criaremos as classes Java correspondentes às entidades de nossa aplicação, os beans ou POJOs. No pacote **br.gov.demoiselle.rh.bean**, criamos as classes Departamento e Funcionario, implementando a interface **IPojo**. A implementação das duas classes deve ficar como na listagem seguinte:

```
package br.gov.demoiselle.rh.bean;

import br.gov.framework.demoiselle.core.bean.IPojo;

@SuppressWarnings("serial")
public class Departamento implements IPojo {
	private long id;
	private String nome;

	public Departamento() {
	}

	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
```

```
        }

        public String getNome() {
                return nome;
        }

        public void setNome(String nome) {
                this.nome = nome;
        }

}

package br.gov.demoiselle.rh.bean;

import br.gov.framework.demoiselle.core.bean.IPojo;

@SuppressWarnings("serial")
public class Funcionario implements IPojo {
        private long id;
        private String nome;
        private Departamento departamento;

        public Funcionario() {
        }

        public long getId() {
                return id;
        }

        public void setId(long id) {
                this.id = id;
        }

        public String getNome() {
                return nome;
        }
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }

    public Departamento getDepartamento() {
        return departamento;
    }

    public void setDepartamento(Departamento departamento) {
        this.departamento = departamento;
    }
}
```

Persistência

Para mapear os beans para as tabelas do nosso banco, temos duas alternativas. Uma é criar arquivos hbm, que são XMLs que descrevem o mapeamento para o Hibernate. Outra é usar anotações e fazer o mapeamento direto na classe. Neste exemplo, iremos utilizar a primeira forma.

Assim, cada bean deve ter seu respectivo arquivo hbm. Vamos criar em **src/main/resources/hbm** os arquivos **Departamento.hbm.xml** e **Funcionario.hbm.xml**. As respectivas listagens são as seguintes:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">
<hibernate-mapping>
    <class name="br.gov.demoiselle.rh.bean.Departamento"
table="departamento">

        <id column="id" name="id">
            <generator class="sequence">
                <param name=
"sequence">departamento_id_seq</param>
            </generator>
```

```
            </id>

            <property column="nome" name="nome"/>

    </class>
</hibernate-mapping>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">
<hibernate-mapping>
      <class name="br.gov.demoiselle.rh.bean.Funcionario"
table="funcionario">

            <id column="id" name="id">
                  <generator class="sequence">
                        <param name=
"sequence">funcionario_id_seq</param>
                  </generator>
            </id>

            <property column="nome" name="nome"/>

            <many-to-one class=
"br.gov.demoiselle.rh.bean.Departamento"
column="id_departamento" name="departamento"/>

    </class>
</hibernate-mapping>
```

Em seguida, é necessário fazer a associação dos hbms ao arquivo hibernate.cfg.xml. Isso é feito pelo menu Demoiselle->Configurar Projeto. Selecionamos a aba Hibernate e preenchemos a tela com os dados de conexão com o banco. Ao clicar no botão Aplicar, o arquivo **hibernate.cfg.xml** é criado. Esse mesmo procedimento serve para alterar o arquivo, caso algum hbm seja incluído ou removido.

Agora criaremos as classes de acesso aos dados, os DAOs. Acessamos o menu Demoiselle->Editar Projeto. Na aba DAO iremos criar duas classes, a partir de nossos dois beans. Então usando o botão Adicionar, criaremos dois DAOS, a partir dos parâmetros da tabela a seguir:

Pacote (br.gov.demoiselle)	Valor (br.gov.demoiselle)	Tipo
.rh.persistence.dao	.rh.bean.Departamento	Hibernate
.rh.persistence.dao	.rh.bean.Funcionario	Hibernate

Esse procedimento gera duas interfaces e duas classes que as implementam. No pacote **br.gov.demoiselle.rh.persistence.dao**: **IDepartamentoDAO** e **IfuncionarioDAO**. No pacote **br.gov.demoiselle.rh.persistence.dao.implementation**: **DepartamentoDAO** e **FuncionarioDAO**.

Vamos testar nossas classes de persistência criando um teste unitário com JUnit. Em src/test/java, criamos um JUnit Test Case chamado TestIncluirDepartamento.

```
package br.gov.demoiselle.rh;

import javax.swing.JOptionPane;

import junit.framework.TestCase;
import br.gov.demoiselle.rh.bean.Departamento;
import br.gov.demoiselle.rh.persistence.dao.IDepartamentoDAO;
import br.gov.framework.demoiselle.core.layer.IBusinessController;
import br.gov.framework.demoiselle.core.layer.integration.Injection;
import br.gov.framework.demoiselle.web.transaction.WebTransaction
Context;

public class TestIncluirDepartamento extends TestCase implements
        IBusinessController {
    @Injection
    IDepartamentoDAO departamentoDAO;
```

```
    public void testIncluir() {
        Departamento departamento = new Departamento();

        assertEquals(0, departamento.getId());

        departamento.setNome(JOptionPane
                .showInputDialog("Digite o nome do
Departamento:"));

        WebTransactionContext.getInstance().init();

        departamentoDAO.insert(departamento);

        WebTransactionContext.getInstance().end();

        assertNotSame(0, departamento.getId());
    }

}
```

Ao rodar esse teste verificamos se ele inclui realmente o registro na tabela de banco de dados.

Filtros de pesquisa

Iremos realizar consultas em nossa pequena base de dados. Para isso, criaremos classes que funcionarão como filtros de pesquisa. Elas ficarão no pacote **br.gov.demoiselle.rh.persistence.dao.filter**. São as classes **FiltroDepartamento** e **FiltroFuncionario**, cujas respectivas listagens vêm adiante:

```
package br.gov.demoiselle.rh.persistence.dao.filter;

import br.gov.component.demoiselle.hibernate.filter.Filter;
import br.gov.demoiselle.rh.bean.Departamento;

@SuppressWarnings("serial")
```

```
public class FiltroDepartamento extends Filter {
        public static final String ID = "id";
        public static final String NOME = "nome";

        public FiltroDepartamento() {
                setClazz(Departamento.class);
        }

}

package br.gov.demoiselle.rh.persistence.dao.filter;

import br.gov.component.demoiselle.hibernate.filter.Filter;
import br.gov.demoiselle.rh.bean.Funcionario;

@SuppressWarnings("serial")
public class FiltroFuncionario extends Filter {
        public static final String ID = "id";
        public static final String NOME = "nome";

        public FiltroFuncionario() {
                setClazz(Funcionario.class);
        }

}
```

Essas classes pertencem ao componente Demoiselle Hibernate Filter, que não faz parte do framework, mas pode ser acoplado a ele. Esse componente não é obrigatório para fazer consultas usando o Demoiselle Framework, mas é muito útil para quem não quer escrever SQL.

Para o utilizarmos teremos que modificar a herança de nossas classes DAO. Em vez delas herdarem de **HibernateGenericDAO**, elas herdarão de **HibernateFilterGenericDAO**. Isso nos dará acesso ao método *find()*, que recebe como argumento um objeto da classe **Filter**.

Com essa modificação, podemos implementar os métodos de busca e listagem para funcionários e departamentos, conforme as listagens a seguir.

```
package br.gov.demoiselle.rh.persistence.dao.implementation;/*
Imports list */

import java.util.List;

import br.gov.component.demoiselle.hibernate.
filter.dao.HibernateFilterGenericDAO;
import br.gov.demoiselle.rh.bean.Departamento;
import br.gov.demoiselle.rh.persistence.dao.IDepartamentoDAO;
import br.gov.demoiselle.rh.persistence.dao.filter.FiltroDepartamento;

public class DepartamentoDAO extends HibernateFilter
GenericDAO<Departamento>
        implements IDepartamentoDAO {
    /* @fwk-methods-begin */

    public Departamento buscarDepartamento(Departamento arg0)
{
        FiltroDepartamento f = new FiltroDepartamento();

        String atributo;
        Object valorDePesquisa;

        if (arg0.getId() > 0) {
            atributo = FiltroDepartamento.ID;
            valorDePesquisa = arg0.getId();
        } else {
            atributo = FiltroDepartamento.NOME;
            valorDePesquisa = arg0.getNome();
        }

        f.addEquals(atributo, valorDePesquisa);

        List<Departamento> results = find(f);

        if (results != null && results.size() > 0) {
            return results.get(0);
```

```
            }
            return null;
        }

        public List<Departamento> listarDepartamentos() {
            FiltroDepartamento f = new FiltroDepartamento();

            List<Departamento> results = find(f);

            if (results != null && results.size() > 0) {
                return results;
            }
            return null;
        }

        /* @fwk-methods-end */
}

package br.gov.demoiselle.rh.persistence.dao.implementation;/*
Imports list */

import java.util.List;

import br.gov.component.demoiselle.hibernate.filter.dao.Hibernate
FilterGenericDAO;
import br.gov.demoiselle.rh.bean.Funcionario;
import br.gov.demoiselle.rh.persistence.dao.IFuncionarioDAO;
import br.gov.demoiselle.rh.persistence.dao.filter.FiltroFuncionario;

public class FuncionarioDAO extends HibernateFilterGenericDAO
<Funcionario>
            implements IFuncionarioDAO {
        /* @fwk-methods-begin */

        public Funcionario buscarFuncionario(Funcionario arg0) {
            FiltroFuncionario f = new FiltroFuncionario();
```

```
            String atributo;
            Object valorDePesquisa;

            if (arg0.getId() > 0) {
                  atributo = FiltroFuncionario.ID;
                  valorDePesquisa = arg0.getId();
            } else {
                  atributo = FiltroFuncionario.NOME;
                  valorDePesquisa = arg0.getNome();
            }

            f.addEquals(atributo, valorDePesquisa);

            List<Funcionario> results = find(f);

            if (results != null && results.size() > 0) {
                  return results.get(0);
            }
            return null;
      }

      public List<Funcionario> listarFuncionarios() {
            FiltroFuncionario f = new FiltroFuncionario();

            List<Funcionario> results = find(f);

            if (results != null && results.size() > 0) {
                  return results;
            }
            return null;
      }

      /* @fwk-methods-end */
}
```

Esses métodos serão utilizados pela próxima camada, a de controle e negócios.

Negócio

Vamos criar um controlador de negócio chamado FuncionarioBC. Para isso, acessamos o menu Demoiselle->Editar Projeto e escolhemos a aba Business Controller. Clicamos no botão adicionar e preenchemos os campos com os valores da tabela seguinte:

Campo	Valor
Nome	FuncionarioBC
Pacote	br.gov.demoiselle.rh.business
DAO	br.gov.demoiselle.rh.persistence.dao.IFuncionarioDAO

A classe **FuncionarioBC** deve ter sido criada no pacote **br.gov.demoiselle.rh.business.implementation**, assim como a interface que ela implementa, **IFuncionarioBC**, foi criada no pacote **br.gov.demoiselle.rh.business**.

Editamos a interface e definimos um método chamado *incluirFuncionario()*, conforme a listagem a seguir.

```
package br.gov.demoiselle.rh.business;/* Imports list */

import br.gov.demoiselle.rh.bean.Funcionario;
import br.gov.framework.demoiselle.core.layer.IBusinessController;

public interface IFuncionarioBC extends IBusinessController {
    /*
     * @fwk-methods-begin
     */
    public void incluirFuncionario(Funcionario arg0);
    /*
     * @fwk-methods-end
     */
```

```
}
```

Isso obrigará a classe FuncionarioBC a implementar o método incluirFuncionário. A implementação ficará como na listagem adiante:

```
package br.gov.demoiselle.rh.business.implementation;/* Imports list
*/

import br.gov.demoiselle.rh.bean.Funcionario;
import br.gov.demoiselle.rh.business.IFuncionarioBC;
import br.gov.demoiselle.rh.persistence.dao.IFuncionarioDAO;
import br.gov.framework.demoiselle.core.layer.integration.Injection;

public class FuncionarioBC implements IFuncionarioBC {
	@Injection
	private IFuncionarioDAO funcionarioDAO;

	/* @fwk-methods-begin */

	public void incluirFuncionario(Funcionario arg0) {
		arg0.setNome(arg0.getNome().toUpperCase());
		arg0.getDepartamento().setNome(

arg0.getDepartamento().getNome().toUpperCase());
		funcionarioDAO.insert(arg0);
	}

	/* @fwk-methods-end */
}
```

Vamos criar um teste unitário para testar a inclusão. Mas antes, vamos adicionar ao Business Controller os outros métodos necessários para a inclusão de um funcionário (listagem 27). Não esqueça de criar esses métodos na interface IFuncionarioBC, caso contrário eles não serão encontrados.

```
package br.gov.demoiselle.rh.business.implementation;/* Imports list
*/
```

```
import br.gov.demoiselle.rh.bean.Departamento;
import br.gov.demoiselle.rh.bean.Funcionario;
import br.gov.demoiselle.rh.business.IFuncionarioBC;
import br.gov.demoiselle.rh.persistence.dao.IDepartamentoDAO;
import br.gov.demoiselle.rh.persistence.dao.IFuncionarioDAO;
import br.gov.framework.demoiselle.core.layer.integration.Injection;

public class FuncionarioBC implements IFuncionarioBC {
	@Injection
	private IFuncionarioDAO funcionarioDAO;
	@Injection
	private IDepartamentoDAO departamentoDAO;

	/* @fwk-methods-begin */

	public void incluirFuncionario(Funcionario arg0) {
		arg0.setNome(arg0.getNome().toUpperCase());
		arg0.getDepartamento().setNome(

arg0.getDepartamento().getNome().toUpperCase());
		funcionarioDAO.insert(arg0);
	}

	public void incluirDepartamento(Departamento arg0) {
		arg0.setNome(arg0.getNome().toUpperCase());
		departamentoDAO.insert(arg0);
	}

	public Departamento buscarDepartamento(Departamento arg0)
{
		return departamentoDAO.buscarDepartamento(arg0);
	}

	public Funcionario buscarFuncionario(Funcionario arg0)
	{
		return funcionarioDAO.buscarFuncionario(arg0);
	}
```

```
        /* @fwk-methods-end */
}
```

Agora criamos no pacote src/test/java a classe TestIncluirFuncionario, conforme a próxima listagem:

```
package br.gov.demoiselle.rh;

import javax.swing.JOptionPane;

import junit.framework.TestCase;
import br.gov.demoiselle.rh.bean.Departamento;
import br.gov.demoiselle.rh.bean.Funcionario;
import br.gov.demoiselle.rh.business.IFuncionarioBC;
import br.gov.framework.demoiselle.core.layer.IBusinessController;
import br.gov.framework.demoiselle.core.layer.integration.Injection;
import br.gov.framework.demoiselle.web.transaction.WebTransactionContext;

public class TestIncluirFuncionario extends TestCase implements
        IBusinessController {
    @Injection
    IFuncionarioBC funcionarioBC;

    public void testIncluir() {
        Funcionario funcionario = new Funcionario();

        assertEquals(0, funcionario.getId());

        funcionario.setNome(JOptionPane
                .showInputDialog("Digite o nome do funcionÃ¡rio:"));

        funcionario.setDepartamento(new Departamento());

        assertEquals(0, funcionario.getDepartamento().getId());

        funcionario.getDepartamento().setNome(
```

```
                    JOptionPane.showInputDialog("Digite o nome
do departamento:").toUpperCase());

            WebTransactionContext.getInstance().init();
            Departamento departamento =
funcionarioBC.buscarDepartamento(funcionario.getDepartamento());
            WebTransactionContext.getInstance().end();

            if (departamento == null)
            {
                  WebTransactionContext.getInstance().init();

funcionarioBC.incluirDepartamento(funcionario.getDepartamento());
                  WebTransactionContext.getInstance().end();
                  departamento =
funcionarioBC.buscarDepartamento(funcionario.getDepartamento());
            }

            funcionario.setDepartamento(departamento);

            WebTransactionContext.getInstance().init();

            funcionarioBC.incluirFuncionario(funcionario);

            WebTransactionContext.getInstance().end();

            assertNotSame(0, funcionario.getId());
            assertNotSame(0, funcionario.getDepartamento().getId());

      }

}
```

Podemos experimentar primeiro incluir funcionários de departamentos existentes e inexistentes. Depois, vamos incluir métodos para listar funcionários e departamentos, conforme próximas listagens, que serão necessários na

próxima camada. Temos que especificar os métodos na interface, senão eles não serão encontrados.

```
public List<Funcionario> listarFuncionarios() {
        return funcionarioDAO.listarFuncionarios();
}

public List<Departamento> listarDepartamentos() {
        return departamentoDAO.listarDepartamentos();
}
```

Apresentação

Vamos inicialmente criar uma regra de navegação para nossa aplicação. Acessamos o menu Demoiselle->Editar Projeto e selecionamos a aba Regras de Navegação.

Criamos uma regra chamada RegraFuncionario no pacote br.gov.demoiselle.rh.constant. Nessa regra um caso de navegação chamado funcionario_listar aponta para a página /private/pages/funcionario_listar.xhtml. Após clicar em Salvar e Aplicar, será criada uma classe chamada AliasNavigationRules, no pacote citado anteriormente. O código é o seguinte:

```
package br.gov.demoiselle.rh.constant;/* Imports list */

public class AliasNavigationRule {
        public static final String ALIAS_FUNCIONARIO_LISTAR =
"funcionario_listar";/*@fwk-methods-begin*//*@fwk-methods-end*/
}
```

Agora vamos criar uma classe chamada FuncionarioMB por meio do Wizard. Acessamos o menu Demoiselle->Editar Projeto e selecionamos a aba Managed Beans. Clique em Adicionar e defina o Managed Bean com os seguintes dados:

Nome	FuncionarioMB
Pacote	br.gov.demoiselle.rh.ui.managedbean
Nome da variável	funcionarioMB
Escopo	session
Business Controller	br.gov.demoiselle.rh.business.IFuncionarioBC

Os POJOs utilizados pelo MB serão Departamento e Funcionario. Definiremos apenas uma ação, chamada listar, cuja ação de retorno é definida pela constante LISTAR_FUNCIONARIOS (gerada pela regra de navegação).

Na classe FuncionarioMB criada, editamos o método *getListFuncionario()*, para obter a listagem do Business Controller:

```
public List<Funcionario> getListFuncionario() {
        this.listFuncionario = funcionarioBC.listarFuncionarios();
        return this.listFuncionario;
}
```

Agora vamos criar a página JSF que irá listar os funcionários cadastrados. No menu Demoiselle, acesse o item Criar páginas. No pacote **/rh/src/main/webapp/private/pages**, vamos criar uma página de listagem. Selecionamos o modelo Listagem e clicamos em Next. Os dados da página são os das tabelas a seguir:

Nome do Arquivo	listafuncionarios.xhtml
Managed Bean	funcionarioMB
Pojo	Funcionario

Os dados das colunas são estes:

Campo	Rótulo
funcionario.id	Id
funcionario.nome	Nome
funcionario.departamento.nome	Departamento

Clicamos em Finish e a página será criada no diretório especificado.

Antes de rodar a página, temos que criar um usuário para fazer a autenticação. No Tomcat, por exemplo, isso é feito no arquivo tomcat-users.xml, conforme segue adiante:

```
<role rolename="funcionario"/>
  <user username="funcionario" password="funcionario"
roles="funcionario"/>
```

Selecionamos o arquivo **listafuncionarios.xhtml** e executamo-lo por meio do menu Run->Run As->Run On Server. Isso sintetiza o desenvolvimento de uma aplicação Web com o Demoiselle Framework.

Conclusão

Demoiselle Framework facilita a padronização das soluções do governo. A padronização se divide em duas frentes, padronização de tecnologias e padronização de arquitetura.

A padronização de tecnologias se dá pela análise, integração e utilização de tecnologias mais reconhecidas utilizadas pelas comunidades de desenvolvedores, com preferência para o uso de padrões abertos.

A padronização de arquitetura, por sua vez, se dá pelo uso de uma interface integradora para os desenvolvedores e por uma arquitetura de referência para as aplicações.

A padronização facilita de suporte e absorção de sistemas, reuso de conceitos e práticas maduras e facilita a integração e disponibilização de serviços para os novos sistemas.

Referências Bibliográficas

[AlCM02] ALUR, Deepak. CRUPI, John. e MALKS, Dan. *Core J2EE: As melhores práticas e estratégias de design*. Campus. Rio de Janeiro, 2002.

[Fowl04] FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. Disponível em <http://martinfowler.com/articles/injection.html>. Acesso em 29/07/2009.

[Fowl05] FOWLER, Martin. *Inversion of Control*. Disponível em <http://martinfowler.com/bliki/InversionOfControl.html>. Acesso em 29/07/2009.

[Fowl06] FOWLER, Martin. *Padrões de Arquitetura de Aplicações Corporativas*. Bookman, Porto Alegre, 2006.

[GHJV00] GAMMA, Erich. HELM, Richard. JOHNSON, Ralph e VLISSIDES, John. *Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos*. Bookman. Porto Alegre, 2000.

[Hors07] HORSTMANN, Cay. *Padrões e Projeto Orientados a Objeto*. 2.ed. Bookman. Porto Alegre, 2007.

[JCP00] JAVA COMMUNITY PROCESS. *JSR-000154 Java Servlet 2.5 Specification*. Disponível em <http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html>. Acesso em 28/07/2009.

[Maci08] MACIAS, Ananda. M. *Frameworks de Desenvolvimento – Visão Geral*. Tematec, Ano X, nº XX, p. 1, 2008. Disponível em <www.serpro.gov.br/clientes/serpro/serpro/imprensa/publicacoes/tematec/2008/ttec92_a>. Acesso em 28/07/2007.

[Mcco05] MCCONNELL, Steve. *Code Complete: Um guia prático para a construção de software*. Bookman. Porto Alegre, 2005.

[Paul09] PAULA FILHO, Wilson. P. *Engenharia de Software: Fundamentos, Métodos e Padrões*. 3.ed. LTC. Rio de Janeiro, 2009.

Formato	*15,5 x 22,5 cm*
Mancha gráfica	*12 x 18,3cm*
Papel	*pólen soft 80g (miolo), duo design 250g (capa)*
Fontes	*Times New Roman 17/20,4 (títulos),*
	12/14 (textos)