Universidade Estadual de Londrina

FERNANDO MIGLIORINI LUIZÃO

UM MECANISMO DE APOIO À CUSTOMIZAÇÃO DE SISTEMAS WEB:

UM ESTUDO DE CASO APLICADO A UM ERP

Londrina
2011

FERNANDO MIGLIORINI LUIZÃO

UM MECANISMO DE APOIO À CUSTOMIZAÇÃO DE SISTEMAS WEB:

UM ESTUDO DE CASO APLICADO A UM ERP

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Orientadora: Profª Dra. Jandira Guenka Palma.

Londrina
2011

Catalogação Elaborada pela Divisão de Processos Técnicos da Biblioteca Central da Universidade Estadual de Londrina

Dados Internacionais de Catalogação-na-Publicação (CIP)

L953m Luizão, Fernando Migliorini.
Um mecanismo de apoio à customização de sistemas *Web* : um estudo de caso aplicado a um ERP / Fernando Migliorini Luizão. – Londrina, 2011.
111 f.: il.

Orientador: Jandira Guenka Palma.
Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2011.
Inclui bibliografia.

1. Engenharia de software – Teses. 2. Sistemas de informação gerencial – Teses. 3. Software – Reutilização – Teses. 4. Interfaces do usuário (Sistemas de computador) – Teses. I. Palma, Jandira Guenka. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519.68.02

FERNANDO MIGLIORINI LUIZÃO

UM MECANISMO DE APOIO À CUSTOMIZAÇÃO DE SISTEMAS **WEB**: UM ESTUDO DE CASO APLICADO A UM ERP

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

_____

Orientadora. Profª Dra. Jandira Guenka Palma
Universidade Estadual de Londrina – UEL

_____

Prof Dr. Evandro Bacarin
Universidade Estadual de Londrina – UEL

_____

Prof Dr. Rodolfo Miranda de Barros
Universidade Estadual de Londrina – UEL

_____

Profª Dra. Elisa Hatsue Moriya Huzita
Universidade Estadual de Maringá – UEM

Londrina, 17 de agosto de 2011.

Em memória de meu pai, que sempre apoiou meus estudos.

AGRADECIMENTOS

À minha esposa pelo apoio, dedicação e paciência. Neste momento, as palavras tornam-se áridas para manifestar tão profunda admiração.

À minha família, sempre presente nos momentos de angústia.

À minha tia, pela ajuda na revisão deste trabalho.

Aos colegas do mestrado, pela convivência e aprendizado.

À minha orientadora, pela ajuda durante a formulação do trabalho, e pelo tempo despendido para nossas conversas.

À empresa que me acolheu, pela oportunidade de aplicar este projeto e avaliar os resultados práticos obtidos.

Aos colegas de trabalho, pelas lições partilhadas.

A todos que de alguma maneira contribuíram para sanar dúvidas pertinentes à pesquisa.

“Penso que há talvez no mundo mercado para cinco computadores.”

(THOMAS WATSON, presidente da IBM, em 1943).

LUIZÃO, Fernando Migliorini. **Um mecanismo de apoio à customização de sistemas web**: um estudo de caso aplicado a um ERP. 2011. 111 f. Dissertação. (Mestrado em Ciências da Computação) – Universidade Estadual de Londrina, 2011.

RESUMO

Com o amadurecimento dos padrões Web, tornou-se possível desenvolver aplicações web quase tão ricas quanto aplicações desktop, de forma que muitos sistemas corporativos tem sido desenvolvidos nesta plataforma. Este tipo de software frequentemente enfrenta necessidades de customizações, e grande parte destas alterações estão localizadas nas interfaces com o usuário. Uma forma comum adotada para lidar com esta necessidade é copiar o código original e fazer as alterações. Entretanto, essa estratégia é inviável, pois manter várias versões de um mesmo sistema torna-se uma tarefa custosa e propensa a erros. Nesta perspectiva, o presente trabalho propõe um mecanismo para simplificar a customização de aplicações web, permitindo a sua composição de acordo com as necessidades. Por meio da estrutura criada, interfaces gráficas são definidas a partir de modelos, e geradas dinamicamente em tempo de execução. O mecanismo permite que modelos sejam combinados de acordo com regras de precedência, possibilitando customizações sem afetar o código original.

Palavras-chave: Customização. Sistema web. Modularidade. Geração de código. ERP.

LUIZÃO, Fernando Migliorini. **A support mechanism for customization of web applications**: a case study applied to an ERP. 2011. 111 p. Dissertation. (Masters in Computer Science) – Universidade Estadual de Londrina, 2011.

ABSTRACT

The maturing of Web standards allowed for the development of web applications almost as rich as desktop applications, so many enterprise systems have been developed on this platform. This kind of software often faces the need for customization, and most of these changes are located in user interfaces. A common way adopted to deal with this need is to copy the original code and make changes. However, this strategy is unfeasible, because maintaining multiple versions of a system becomes costly and error prone. In this perspective, this paper proposes a mechanism to simplify the customization of web applications, allowing the composition according to the needs. Through the Framework established, graphical interfaces are defined using templates, and dynamically generated at runtime. The mechanism allows them to be combined according to precedence rules, allowing customization without affecting the original code.

Keywords: Customization. Web applications. Modularity. Code generation. ERP.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BO	Business Object
CPF	Cadastro de Pessoa Física
CRUD	Create, Retrieve, Update, Delete
CSS	Cascading Style Sheets
DHTML	Dynamic HTML
DFD	Diagrama de fluxo de dados
DSL	Domain Specific Language
DTE	Diagrama de transição de estados
DOM	Document Object Model
EBT	Enduring Business Theme
ERP	Enterprise Resource Planning
GIT	Global Information Tracker
GUI	Graphical User Interface
HTML	Hypertext Markup Language
JSON	Javascript Object Notation
MDD	Model Driven Development
MVC	Model-View-Controller
OCP	Open/Closed Principle
RDF	Resource Description Framework
RIA	Rich Internet Applications
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Location
XBL	XML Bindings Language
XML	Extensible Markup Language
XPFE	Cross-Plataform Front End
XPI	Cross-Platform Installer
XUL	XML User Interface Language
YAML	YAML Ain't Markup Language
W3C	World Wide Web Consortium

SUMÁRIO

1 INTRODUÇÃO

Com o amadurecimento dos padrões Web e a evolução dos navegadores, tornou-se possível desenvolver aplicações web quase tão ricas quanto aplicações desktop. Aliada à facilidade de atualização e à possibilidade de acessar estes sistemas pela internet usando apenas um navegador, essa plataforma mostra-se interessante para o desenvolvimento de sistemas de gestão empresarial, por permitir o acesso a informações atualizadas de qualquer lugar do mundo, inclusive a partir de outros tipos de equipamentos com poucos recursos computacionais, como telefones celulares.

Enterprise Resource Planning (ERP) é um tipo de software bastante complexo que visa atender a todas as necessidades de uma organização em se tratando de sistemas de informação, integrando diferentes departamentos e cobrindo uma ampla gama de processos de negócio. Oferecem informações atualizadas, auxiliando os gestores durante a tomada de decisões, permitindo reduzir custos, fazer um planejamento mais preciso, com expectativas realistas e objetivos mais claros, e traçar estratégias mais competitivas.

Como os ERPs tendem a abranger toda a empresa, esse tipo de software possui escopo mais amplo que projetos de desenvolvimento tradicionais, os quais frequentemente focam em um ou mais departamentos ou processos de negócio e são destinados a aplicações específicas. Os riscos associados são relativamente maiores, ressaltam Luo e Strong (2004), ao explicar que a exigência de integração de dados, processos e operações de toda a empresa torna os projetos muito mais complexos.

Além disso, por visar atender necessidades de nichos de negócios diferentes, esse tipo de software vem com algumas pressuposições embutidas em relação aos processos da organização, as quais raramente correspondem exatamente aos processos já existentes. Como consequência, Luo e Strong (2004) afirmam que a adequação entre os processos da empresa e o sistema de gestão é crítica para o sucesso do projeto, e Nordheim e Paivarinta (2004) reforçam dizendo que customização mínima do software tem sido considerada um fator crítico de sucesso para implantações.

Entretanto, segundo Rothenberg e Srite (2009), atualizar projetos customizados é trabalhoso porque as customizações devem ser realinhadas ao sistema atualizado ou mesmo reescritas. Nesta perspectiva, torna-se necessário estudar estratégias para evoluir o sistema e permitir customizações de forma a reduzir o impacto na manutenção e nos custos de desenvolvimento.

1.1 MOTIVAÇÃO

A realização deste estudo foi motivada pela observação acerca do desenvolvimento de um ERP por uma pequena empresa de Londrina-PR, que tem enfrentado diversas dificuldades. Os principais problemas encontrados durante o desenvolvimento estão relacionados a falhas na gerência de configuração e versionamento do projeto.

O primeiro está relacionado à complexidade inerente aos ERPs, decorrente do seu elevado número de funcionalidades, e também dos relacionamentos entre elas. Muitas funcionalidades precisam trocar informações para desempenhar seu papel corretamente, o que muitas vezes resulta em alto acoplamento no código-fonte, que se torna difícil de manter, exigindo muito tempo e esforço. Além disso, esta forte dependência também dificulta a separação de funcionalidades e impede a venda avulsa de módulos. Como parte da estratégia comercial, a empresa pretende vender módulos separadamente, caso o cliente tenha necessidade de apenas um subconjunto do software. Atualmente, o sistema é desenvolvido de forma monolítica, não existindo uma separação efetiva dos módulos.

Um segundo problema diz respeito à maneira adotada para lidar com a necessidade de customizações, que se provou custosa e propensa a erros, além de dificultar o versionamento do software. Foi constatado que, no modelo de desenvolvimento da empresa, cada customização gera uma nova versão do sistema, específica para cada cliente, que é mantida paralelamente ao código principal na forma de ramificações (branches) no sistema de controle de versões. Esta abordagem tem causado muitos problemas e retrabalho, pois conforme o número de clientes aumenta, a tarefa de manutenção torna-se mais difícil. Essa estratégia é inviável mesmo em curto prazo, pois corrigir um defeito que afeta todas as versões, por exemplo, implica em aplicar a mesma correção a todas as versões diferentes, e levando em consideração as customizações efetuadas.

A realização da presente pesquisa é sustentada ainda pela importância que a tecnologia de informação vem assumindo na estratégia de empresas, remetendo à necessidade de repensar a forma como se desenvolve software, para facilitar a manutenção. Esta abordagem faz-se necessária, devido ao fato de os modelos de desenvolvimento de software não terem mudado muito nas últimas décadas (VALKOV, 2008).

1.2 Objetivos

O objetivo geral desta pesquisa consiste em moldar um mecanismo que simplifique a customização de sistemas baseados em web, de modo que seja possível adicionar, remover e modificar funcionalidades visando atender as necessidades únicas de cada organização, sem apresentar os problemas citados. Deve permitir a composição do sistema de acordo com as necessidades por meio da instalação de módulos, que se integram facilmente ao núcleo do software. Além disso, deve possuir a capacidade de aplicar customizações separadamente ao núcleo do sistema por meio de módulos, tornando possível fazer alterações mantendo o código customizado separado do original, permitindo que sejam facilmente aplicadas e removidas.

As principais customizações abordadas neste estudo estão relacionadas às interfaces com o usuário. Foi constatado, observando os registros da empresa analisada, que elas são os aspectos que mais mudam, possivelmente por ser a porção visível da aplicação, em que o usuário interage diretamente. Dentro desta categoria enquadram-se alterações em cadastros e relatórios, bem como desenvolvimento de novas funcionalidades de ambos os tipos. Também são abordadas customizações no comportamento do sistema, entretanto, em menor grau, visto que nos casos analisados, não houve necessidade de customizações agressivas nas regras de negócio.

Para simplificar a customização das interfaces, foi criado um modelo para descrição das mesmas, de forma que sejam combinadas pelo mecanismo para fazer customizações, sendo assim geradas em tempo de execução. É esperado que esta abordagem reduza os problemas de proliferação de versões relacionados à adaptação aos processos da empresa, separando o código customizado do original e fornecendo mais segurança aos programadores. Também é esperado que o grau de abstração adicionado pelos modelos propostos reduza o tempo e os custos necessários para as customizações mais comuns, minimize a quantidade de defeitos e aumente a produtividade da equipe.

1.3 Organização da Dissertação

O presente trabalho encontra-se estruturado em cinco capítulos. Após apresentar a motivação e os objetivos do estudo, o segundo capítulo traz o referencial teórico sobre sistemas web, ERPs, customização de softwares, frameworks, bem como as técnicas relacionadas a ambientes computacionais dinâmicos.

O terceiro capítulo apresenta a proposta de um mecanismo que visa facilitar a customização de sistemas web. Detalha-se sua estrutura de módulos, como eles devem ser integrados e os modelos que foram criados visando simplificar o processo de customização.

No quarto capítulo é descrita a aplicação do mecanismo em um estudo de caso realizado em uma empresa do município de Londrina-PR, visando avaliar a viabilidade da proposta em um projeto real de ERP. Por fim, são tecidas as considerações finais, bem como as sugestões para trabalho futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma visão geral acerca da recente tendência de migração dos softwares para a Web. Será visto como esta plataforma é interessante para o desenvolvimento de aplicações corporativas, na qual se enquadram ERPs, que frequentemente necessitam de customizações. Serão vistas as causas que levam customização em excesso e os impactos causados. Finalmente, serão vistas as bases teóricas acerca de frameworks, geração de código e reflexão computacional, que se mostram eficientes quando é necessário lidar com requisitos voláteis.

2.1 SISTEMAS WEB

Uma das maiores mudanças para aplicações corporativas nos últimos anos tem sido o aumento das aplicações web, pois elas trazem consigo uma série de vantagens: o único software a ser instalado é um navegador; uma abordagem da interface do usuário comum, e acesso universal fácil. Além disso, muitos ambientes tornam mais fácil construir uma aplicação web.

Os navegadores permitem que as pessoas utilizem qualquer tipo de computador para acesso a aplicações pela Internet. Carneiro Júnior et. al. (2007) explicitam que as aplicações Web estão ganhando cada vez mais importância, conforme as conexões ficam mais rápidas e a adoção de banda larga aumenta, os softwares baseados em web vêm substituindo os distribuídos de maneira tradicional.

Este tipo de sistema proporciona maior comodidade aos usuários, uma vez que o software baseado na Web funciona em qualquer plataforma que suporte um navegador, e não há necessidade de instalar ou fazer o download, podendo ser acessado de qualquer lugar, pela Internet. Devido à facilidade de distribuição, o ritmo das mudanças no mercado de software baseado na web é rápido. Ao contrário do software tradicional, que deve ser instalado individualmente em cada computador, mudanças nas aplicações web podem ser entregues rapidamente e recursos podem ser adicionados de forma incremental. Quando uma correção é efetuada em um aplicativo Web, todos os usuários obtêm o benefício ao mesmo tempo, normalmente sem qualquer interrupção no serviço, imprimindo a garantia de qualidade ao produto. A economia obtida com a embalagem e distribuição pode ser reinvestida em tempo de desenvolvimento, gerando qualidade e inovação (CARNEIRO JÚNIOR et al, 2007).

As vantagens do software baseado na Web incluem a facilidade de distribuição, implementação e manutenção, independência de plataforma e maior acessibilidade. Em uma plataforma grande como é a web, existem restrições, como a falta de compatibilidade entre os navegadores, a falta de uniformidade que torna difícil a criação de aplicativos, além da complexidade inerente a uma aplicação web típica (CARNEIRO JÚNIOR et. al., 2007).

À época de seu surgimento, a web tinha como objetivo a troca de conteúdos por meio de páginas HTML (Hypertext Markup Language) estáticas, acessadas pelos navegadores. Porém, quando se percebeu o seu potencial de comunicação e interação, foi constatado que páginas estáticas não seriam suficientes, havendo a necessidade de servir páginas HTML geradas dinamicamente baseadas nas requisições dos usuários. Desta forma, as aplicações web têm evoluído a partir do modelo cliente-servidor para aplicações ricas e interativas, conhecidas como RIA (Rich Internet Applications).

Fraternali et. al, (2010) explicitam que as RIA introduziram novas funcionalidades para a arquitetura web, provenientes de aplicativos cliente-servidor e desktop, as quais permitem melhorar a experiência do usuário. O termo, segundo Bozzon et. al. (2006), denota uma unificação das aplicações desktop e web, visando alavancar as vantagens e superar as desvantagens de ambas as arquiteturas; RIA fornecem interfaces sofisticadas para representar processos e dados complexos, minimizando transferências de dados entre cliente e servidor e um deslocamento das camadas de interação e apresentação do servidor para o cliente. A tabela 1 compara as características entre aplicações desktop, web convencional, e RIA.

Tabela 1 – Comparação de aplicações Desktop, Web e RIA

Característica	Cliente/Servidor, Desktop	Web convencional	RIA
Cliente Universal	Não	Sim	Sim
Instalação do cliente	Complexa	Simples	Simples
Capacidade de interação	Rica	Limitada	Rica
Regra de negócio no servidor	Sim	Sim	Sim
Regra de negócio no cliente	Sim	Limitada	Sim
Necessário recarregar toda a página	Não	Sim	Não
Consultas frequentes ao servidor	Não	Sim	Não
Comunicação do servidor com cliente	Sim	Não	Sim
Funcionamento desconectado	Sim	Não	Sim

Fonte: Bozzon et.al. (2006)

Os benefícios das RIA, observáveis na tabela apresentada, são relevantes para uma variedade de tarefas, como a classificação de dados, filtragem, visual complexo, processamento etc. Normalmente, uma aplicação rica é carregada pelo cliente, juntamente com alguns dados iniciais, em seguida, gerencia os dados e processamento de eventos, comunicando-se com o servidor apenas quando houver necessidade de mais informações (BOZZON et.al., 2006).

As características comumente observadas em aplicações desse tipo podem ser assim elencadas: a) devem suportar o processamento pelo cliente e reduzir a comunicação com o servidor para o mínimo; b) os dados de suporte das aplicações podem ser armazenados com níveis diferentes de persistência, tanto no cliente quanto no servidor, c) processamento de dados, a criação, modificação e filtragem de dados podem ocorrer tanto no cliente quanto no servidor; d) o uso on-line e off-line deve ser possível (BOZZON et.al., 2006).

Segundo Bozzon et. al. (2006), várias tecnologias foram propostas para suportar o desenvolvimento de aplicações ricas. No presente trabalho é considerada a utilização das linguagens HTML e CSS (Cascading Style Sheets), que compõem a interface da aplicação, e JavaScript, responsável pela implementação da lógica no cliente, ou seja, o comportamento da interface. Goodman (1998) refere que a combinação dessas tecnologias permite a criação de páginas web dinâmicas, um amálgama conhecido como o DHTML (Dynamic HTML). Elas são padrões maduros e amplamente usados, definidos pelo W3C

(World Wide Web Consortium), cuja responsabilidade é padronizar qualquer tecnologia relacionada à Web. Cada um destes padrões possui uma responsabilidade na estruturação de uma aplicação. O HTML é responsável pela organização da informação; o CSS, pela formatação e o Javascript, pelo comportamento.

Para Goodman (1998), o propósito do HTML é prover significado estrutural para o conteúdo das páginas. Por sua vez, Silva (2008) acrescenta que esses documentos podem ser lidos por softwares chamados de agentes de usuário, como um navegador, um leitor de tela ou um robô de busca. O HTML atualmente compreende 91 elementos, alguns dos quais em desuso, que visam adicionar semântica ao conteúdo; a linguagem serve exclusivamente para marcação e estruturação de conteúdos.

Uma folha de estilo CSS define como o conteúdo deve ser renderizado na página. Segundo Schmitt (2009), CSS é um sistema simples e padronizado, que proporciona um amplo controle sobre a apresentação das suas páginas web. Feldt (2007) explica que se trata de um mecanismo para adicionar estilo (cores e dimensões) a elementos, e torna possível separar conteúdo da formatação, possibilitando alterar praticamente todos os atributos de um elemento com uma alteração na folha de estilo. O autor complementa que CSS é uma ferramenta poderosa para customização, por possuir noções fortes de precedência.

A linguagem JavaScript, por sua vez, oferece todas as características necessárias a uma linguagem de programação, como tipos de dados, variáveis, estruturas de controle etc. Segundo Flanagan (2006), ela é interpretada pelo navegador em tempo de execução, e permite que conteúdo executável seja distribuído pela Internet. Por sua natureza dinâmica, possibilita interações avançadas com o navegador que de outra forma não seriam possíveis.

Essas características mostram-se interessantes para o desenvolvimento de aplicações empresariais uma vez que permitem o acesso a informações atualizadas, mesmo sem grandes recursos computacionais. ERPs encaixam-se neste perfil de utilização por um número cada vez mais crescente de usuários.

2.2 ERP - Enterprise Resource Planning

Conforme mencionado na introdução deste estudo, a utilização de ERPs por um grande número de empresas parte da necessidade de apoiar as principais funções da organização, incluindo vendas, distribuição, finanças, produção e gestão de recursos humanos,

com vistas a integrar as funcionalidades dentro de um software único, em que todos os módulos podem trocar informações rapidamente.

Para Oram e Wilson (2007), ERP é um software que tem como objetivo integrar todos os dados e processos de uma organização. Como este é um verdadeiro desafio, a indústria de ERP oferece diferentes versões do mesmo software de ERP para diversos segmentos econômicos, tais como petróleo e gás, mecânica, automobilística, farmacêutica e do governo.

Os autores apontam que software ERP geralmente consiste em uma série de módulos que automatizam as operações da organização. Os módulos mais comuns incluem finanças, controle de estoque, folha de pagamento, planejamento e controle da produção, vendas e contabilidade. Estes módulos são projetados para personalização e adaptação, porque, embora as organizações de um determinado segmento econômico utilizem certas práticas comuns, cada organização precisa adaptar o sistema de ERP para suas necessidades específicas. Software ERP também evolui rapidamente para acompanhar a evolução dos negócios que sustenta, e mais módulos são adicionados ao longo do tempo (ORAM; WILSON, 2007).

Oram e Wilson (2007) sinalizam que ser flexível significa ser adaptável a vários modelos de negócios, sem incorrer em custos elevados para as alterações e manutenção. Para isso, é necessário definir um núcleo a partir do qual os novos componentes podem ser facilmente obtidos para fins específicos. Este modelo deve ser abstrato o suficiente para abraçar todos os conceitos básicos do negócio.

De acordo com Arinze e Anandarajan (2003), a maioria das empresas de grande porte utiliza ERPs para apoiar as principais funções da organização, incluindo vendas, distribuição, finanças, produção e gestão de recursos humanos. O apelo de um software desse tipo é a ampla integração de funcionalidades dentro de um pacote único, em que todos os módulos podem trocar informações rapidamente. Segundo Hau e Aparício (2008) a vantagem em ter uma única plataforma é obter a informação coerente e consistente em toda a organização e Luo e Strong (2004) completam citando outros benefícios como substituição de sistemas legados fragmentados, e adoção das melhores práticas nos processos organizacionais.

Rothenberg e Srite (2009) salientam que um processo de negócio é definido como um conjunto de tarefas relacionadas que usam recursos de uma organização para atingir algum resultado. Este consiste em tarefas, recursos, resultados, e relacionamentos entre tarefas e recursos.

Para tanto, segundo Hau e Aparício (2008), uma implantação típica de ERP tem as seguintes fases: análise de lacunas (gap analysis) para definir processos de negócio e a necessidade de customizacão, design e desenvolvimento do sistema; conversões de dados de sistemas legados; testes do novo sistema e treinamento dos usuários.

Na análise de lacunas, são definidas as especificidades da empresa e as possibilidades de alteração ou manutenção de estratégias utilizadas no sentido de manter ou melhorar sua vantagem competitiva. Na fase posterior, são analisadas as necessidades não atendidas pelo sistema a ser implantado e ocorre um processo de modelagem que tem como objetivo o desenvolvimento de novas funcionalidades para supri-las. Outra etapa da implantação consiste no aproveitamento de informações já existentes em sistemas legados da empresa e consequente utilização no novo sistema. Este, então, é testado e a fase final da implantação destina-se ao treinamento dos usuários no intuito de prevenir erros de utilização que podem comprometer a viabilidade do projeto.

Valkov (2008) acrescenta que existem vários problemas não resolvidos no que diz respeito ao desenvolvimento de ERP comerciais, dentre os quais destaca a complexidade e dificuldades de ampliação e atualização, fatores que desencadeiam altos custos, bem como grandes esforços durante o desenvolvimento. Em particular, afirma que a integração e implementação são muito complicadas, lentas, custosas e nem sempre atendem totalmente as necessidades do cliente. Ademais, ressalta que as interfaces com o usuário são frequentemente inflexíveis e não customizáveis devido ao fato de o software se propor a resolver muitos problemas de uma maneira não modular. Nem sempre a interface é adaptável, posto que frequentemente depende das decisões do programador, e não das preferências do usuário.

Considerando que este tipo de sistema requer um mínimo de customização para atender necessidades de nichos específicos de negócios, é importante aprofundar os motivos que levam à customização e seus problemas recorrentes.

2.2.1 Customização de ERP

No cenário empresarial contemporâneo, os sistemas estão ficando cada vez mais complexos, aumentando custos de desenvolvimento e provocando dificuldade de manutenção e evolução. Frequentemente integrar uma funcionalidade não prevista requer um profundo conhecimento da estrutura interna do software e os impactos que possam ser

ocasionados pelas alterações. Isso consome tempo e é extremamente propenso a introduzir erros ao software.

Conforme afirmam Johnson e Foote (2008), o desenvolvimento de novos sistemas é caro, e mantê-los é ainda mais oneroso. Segundo os autores, entre 60 a 85% do custo total do software é devido à manutenção, que pode ser categorizada como corretiva, adaptativa e aperfeiçoamento. Manutenção corretiva é o processo de diagnosticar e corrigir erros, enquanto a adaptativa consiste em atividades necessárias para integrar corretamente um software com um novo hardware, periféricos etc. Aperfeiçoamento é necessário quando um software é bem sucedido, pois conforme um produto é usado, é exercida pressão para melhorar e ampliar suas funcionalidades; essa atividade corresponde a 60% de toda a manutenção, enquanto a manutenção corretiva e adaptativa correspondem à cerca de 20% cada uma.

Considerando que 60% da atividade de manutenção é aperfeiçoamento, uma fase evolutiva é uma parte importante do ciclo de vida de um produto de software bem sucedido. Em muitos aspectos, a manutenção é uma tarefa mais exigente do desenvolvimento. O mantenedor deve interpretar um projeto complexo, mesmo que frequentemente a documentação interna seja inadequada ou inexistente, e fazer as modificações (FOOTE, 1988).

Fayad et.al. (2000) explicam que defeitos de software e deterioração são causados por alterações. Muitas dessas mudanças não podem ser evitadas, entretanto seus efeitos podem ser minimizados. Quando as mudanças devem ser feitas em um software, seja pela inserção de uma nova tecnologia ou por diversificação da clientela, muitas vezes é necessário que todo o programa seja reestruturado. Os autores consideram este processo de reengenharia inadmissível, pois uma vez que o objetivo principal do produto de software não tenha sido alterado, não há razão para que todo o projeto seja redesenhado para incorporar uma mudança.

Yau et. al. (2006) afirmam que enfrentar a customização em uma ampla gama de domínios de aplicação é altamente desejável, mas muito desafiador, porque normalmente há um número de diferentes maneiras de personalizar um componente. Domínios de aplicação peculiares podem exigir personalizações muito diferentes sobre o mesmo componente. Sob este viés, torna-se interessante estudar maneiras de evitar reestruturações desnecessárias e dispendiosas, permitindo que o software seja reconfigurado para atender necessidades específicas sem alterações em sua estrutura básica. Para Talevski et al. (2004), em decorrência da complexidade do software em larga escala e rápida evolução das

necessidades organizacionais, a importância da reconfiguração de software é crescente. A customização emerge como uma resposta efetiva para este problema.

Diferentes autores têm apresentado conceitos de customização. Nordheim e Paivarinta (2004) definem customização como a atividade que consiste em modificar as propriedades de um software, de forma que o sistema resultante se aproxime dos requisitos da organização a que se destina. Os autores empregam a customização no sentido de modificar o software antes do uso. Para Rothenberg e Srite (2009) a customização implica a criação de funcionalidades específicas, alterar código e/ou incluir pacotes de terceiros que necessitem de algum grau de programação para implementar.

Dittrich e Vancouleur (2008) sugerem que o termo customização abrange uma gama de diferentes alterações na aplicação:

a) As tarefas mais simples são as customizações dos relatórios existentes. Muitas vezes, os relatórios personalizados precisam fornecer informações adicionais.
b) A funcionalidade existente pode ser melhorada: uma melhoria simples seria a captura de dados adicionais para alguma entidade, exigindo a mudança em um formulário e em algumas unidades de código, bem como a extensão de uma tabela de banco de dados.
c) Funcionalidades complementares formam uma terceira categoria, composta por recursos mais independentes. Para atender necessidades únicas deve ser possível adicionar outras funcionalidades. Isso exige desenvolvimento de código adicional, mas não requer modificação no código existente.

Para Dittrich e Vancouleur (2008), com a customização, é possível efetuar alterações em uma funcionalidade que se tornam necessárias quando a flexibilidade oferecida pela configuração não é suficiente. O objetivo de ambas é permitir personalizações em vez oferecer um grande número de funções não utilizadas. Parnas (1979) reforça que alguns usuários podem requerer apenas um subconjunto dos serviços ou funcionalidades que outros necessitam, recusando-se a pagar pelos recursos desnecessários.

A partir dessa reflexão, observa-se que a necessidade de customização é reforçada pelo anseio de atender as expectativas de negócios diferentes. Como cada tipo de negócio tem suas particularidades, os clientes muitas vezes precisam que o sistema seja adaptado para melhor atender a sua demanda.

De acordo com Rothenberg e Srite (2009), em todas as instalações de ERP, é necessário algum grau de customização, pois mesmo aplicações projetadas para funcionar em organizações diferentes frequentemente não oferecem todas as funcionalidades necessárias para um nicho específico. Um estudo apontado por Rothenberg e Srite (2009), que analisou 86 organizações, confirmou a importância de um mínimo de customização para o sucesso da implantação de um ERP. Os resultados do estudo enfatizam também que é necessário que processos de negócio sejam realinhados de acordo com as especificidades de cada empresa. Para Luo e Strong (2004), a principal meta da customização é adequar o sistema e os processos que o sistema suporta.

Contudo, ao se reportar a uma pesquisa realizada por Themistocleous et al. (2001) entre profissionais de ERP, Dor et al (2008) destacam que 72% responderam que problemas de customização são um problema técnico sério, que ocupa o segundo lugar entre os maiores problemas técnicos, seguido de integração com sistemas existentes. No entanto, Nordheim e Paivarinta (2004) afirmam que a customização não necessariamente implica em adaptação total do software às necessidades de uma organização. Em alguns casos, a organização pode se adaptar ao software. Em implantações de ERP, isso é um fenômeno bem conhecido, referido como adaptação mútua.

Luo e Strong (2004) argumentam que a implantação de um ERP é frequentemente acompanhada de muitas mudanças organizacionais. Consequentemente, uma questão chave na implantação é como encontrar o ponto ideal entre o sistema e os processos da organização, customizando tanto o sistema quanto a organização. Afirmam que tanto o sistema quanto o processo podem ser alterados ou customizados para alcançar essa meta. Referem que quando o sistema é customizado para adequar-se ao processo já existente na empresa, trata-se de customização técnica. De forma similar, quando um processo é customizado para adequar-se ao sistema, ocorre a customização do processo.

Davenport (apud LUO; STRONG, 2004) elencou três opções de customização técnica: seleção de módulos, configuração e modificação de código, e explica que na seleção de módulos, a empresa escolhe implementar um ou mais módulos usando as configurações padrão definidas pelo fornecedor do ERP. Neste caso, a customização é alcançada por meio da decisão de quais módulos implantar e raramente é suficiente.

Também é possível customizar por meio da seleção de opções de configuração para atender as necessidades da organização e avaliar as consequências de cada opção. Como existem muitas opções em um ERP, essa pode ser uma tarefa complexa e que consome tempo, especialmente quando as interdependências entre opções de módulos

precisam ser consideradas. Os benefícios desse tipo de customização incluem a habilidade de alterar o sistema sem código, suporte total do fabricante e facilidade em futuras atualizações (LUO; STRONG, 2004). Entretanto, no entendimento de Yau et.al. (2006), o principal problema dessa abordagem é que as opções de customização são pré-determinadas e, consequentemente, limitadas.

O terceiro tipo é modificação de código, em que o código fonte é alterado, a funcionalidade é expandida, ou uma nova interface é adicionada. Este tipo de customização oferece maior flexibilidade em adaptar o sistema às necessidades organizacionais e permite integrar o ERP a quaisquer sistemas existentes. Por outro lado, apresenta maior risco, custo mais elevado e probabilidade de falhas. Além disso, o excesso de customização pode levar à incompatibilidade com novas versões do sistema e, portanto, dificultar futuras atualizações. Conforme se evolui de seleção de módulos para customização de código, os riscos aumentam e os benefícios podem não ser proporcionais (LUO; STRONG, 2004).

Rothenberg e Srite (2009) explicitam que, em caso de uma atualização, a funcionalidade pode ser mantida se os desenvolvedores do novo módulo respeitarem os padrões do ERP. Em último caso, o código do ERP pode ser alterado para se adequar às necessidades da organização. Isso exige um esforço substancial de desenvovimento e é problemático, pois a modificação pode ter que ser refeita quando o sistema for atualizado. De acordo com os mesmos autores, customizações que envolvem adições extensivas ou modificações no código do sistema podem comprometer o sucesso de um projeto, uma vez que o excesso de customização aumenta os custos e limita a capacidade de manutenção do sistema.

Face à observação dos pressupostos apresentados, faz-se necessária a investigação dos motivos que levam à necessidade premente de customização. Muitas vezes é possível adequar os processos já existentes na organização em vez de promover alterações no sistema. Isto é conhecido como customização do processo. Rothenberg e Srite (2009) classificam este tipo de customização em três categorias, com base nas mudanças feitas nesses elementos: sem mudança, mudança incremental, mudança radical. A quantidade de mudanças depende de quão bem a nova tecnologia se adequa ao processo existente. A gerência tem a opção de mudar o processo para adequar ao sistema e vice-versa.

Portanto, Rothenberg e Srite (2009) enfatizam que é necessário decidir quando serão feitas customizações no sistema e quando serão feitas alterações nos processos de negócio. Uma adoção bem sucedida deve alinhar software e processos de negócio. Considerar os processos existentes e manter customizações a um mínimo é importante. Uma

organização deve estar apta a implementar as práticas suportadas pelo ERP tanto quanto possível para minimizar customizações. No entanto, para Arinze e Anandarajan (2003), se os processos originais da organização lhe permitiram ganhar uma vantagem competitiva em sua indústria ou são mais adequados à sua cultura, ao forçar a empresa a adotar o processo do sistema, a vantagem é perdida. Na tabela 2, é apresentado um roteiro para decidir qual opção de customização usar, de acordo com o grau de maturidade da empresa.

Tabela 2 – Possibilidades de customização

OPÇÕES DE CUSTOMIZAÇÃO DE PROCESSO				
OPÇÕES DE CUSTOMIZAÇÃO TÉCNICA		Sem mudança	Mudança incremental	Mudança radical
	Customização de módulo	Sem customização Processos de negócio adequados aos processos do sistema	Adaptação de processo Processo do sistema é ideal e processo de negócio está próximo	Conversão do processo Processo do sistema é ideal e o processo de negócio está distante
	Configuração	Adequar sistema ao processo É desnecessário mudar os processos de negócio	Adaptação mútua Processos do sistema e de negócios são próximos, modificações mínimas em ambos	Adequar processo ao sistema Pequenas mudanças no sistema, alinhar processos de negócio com processos do sistema
	Customização de código	Conversão do sistema Não é desejado alterar os processos de negócio Customizar os processos do sistema aos processos de negócio	Conversão do sistema e adaptação do processo Pequenas alterações no processo são desejáveis Customizar os processos do sistema aos processos de negócio	Reengenharia do processo e do sistema Reestruturação total dos processos de negócio e do sistema

Fonte: (LUO; STRONG, 2004, p. 326)

A tabela 2 apresenta nove opções de customização que dependem de quanto mudar os processos e quanto mudar o sistema, tendo em vista as variáveis que implicam na profundidade das mudanças que irão afetar a empresa e o próprio sistema. Quanto mais madura a empresa, menor é a probabilidade de que seja necessário reestruturar seus processos; na maior parte das vezes as melhores práticas do ERP são as mesmas que a organização já utiliza.

A opção pelo tipo de customização a ser adotada mostra-se imprescindível, uma vez que na prática, verifica-se que a customização técnica pode ocasionar vários problemas. A este respeito, Rothenberg e Srite (2009) relacionam alguns fatores que causam o alto grau de customização. Em primeiro plano, a inexperiência da equipe responsável pela implantação pode ocasionar o desenvolvimento de funcionalidades customizadas já disponíveis no sistema padrão, se os membros não estiverem cientes da funcionalidade durante a implantação. Uma equipe com conhecimento limitado sobre os processos da organização pode manter muitos processos de negócios existentes porque eles parecem imutáveis. Além disso, a equipe pode obter menos sucesso em limitar customizações apenas ao necessário.

Outro fator refere-se à resistência a mudanças, pois a implantação dos processos de um ERP normalmente requer suporte dos departamentos operacionais de uma organização, porque envolve mudanças nos processos de negócio existentes. Uma organização resistente a mudanças frequentemente mostra-se relutante em relação a mudar até processos simples se eles forem diferentes das melhores práticas do ERP, mesmo quando os processos podem ser alterados sem perda de funcionalidade operacional.

Pouca aceitação do projeto, questões culturais individuais e organizacionais podem levar à resistência a mudar processos. Isso pode conduzir a customizações desnecessárias. Se houver resistência em relação a alterar processos, a equipe pode não ser capaz de convencer os tomadores de decisão de que é importante manter o mínimo de modificações do sistema. Como resultado, o time de implantação irá customizar o ERP, mesmo em situações em que reorganizar processos de negócio seria uma escolha melhor.

Visando endereçar o problema dos requisitos voláteis, os frameworks situam-se como domínios de aplicação que são caracterizados por rápida mudança de requisitos de software e constituem desafios que são muito diferentes dos que devem ser abordados em domínios mais convencionais (FOOTE, 1988). Uma maneira pela qual a orientação a objetos ajuda a atingir esta meta é estimular o design de componentes gerais, bibliotecas de aplicação independentes e reutilizáveis. O surgimento de um framework

específico para a aplicação em face dos requisitos voláteis é talvez a consequência mais interessante do uso de técnicas orientadas a objeto. Em geral, o uso de frameworks tem contribuído para o desenvolvimento de várias aplicações complexas (CORRÊA, 1997). Este tema será explorado a seguir.

2.3 FRAMEWORKS

Os desafios associados a ambientes de aplicação caracterizados pela rápida evolução dos requisitos não são bem abordados pelas técnicas tradicionais de engenharia de software. O uso de frameworks orientados a objeto preenche esta grande lacuna, e fornece um caminho ao longo das quais as funcionalidades podem evoluir à medida que amadurecem. frameworks apresentam uma maneira de generalizar uma aplicação para satisfazer as exigências distintas, permitindo que versões derivadas divirjam gradualmente do projeto original (FOOTE, 1988).

De acordo com Foote (1988) o projeto de um software não deve ser visto como uma fase, que ocorre no início do ciclo de vida do software, mas como um processo que o permeia. O autor alerta para o risco de se colocar todo o esforço de projeto no início do mesmo, pois pode se tornar difícil de abandonar em estágios avançados, mesmo quando uma solução melhor é identificada mais tarde no processo de desenvolvimento do produto. Sob esse prisma, é possível resolver o problema atual, sabendo que futuramente os componentes resultantes poderão servir como a raiz de um framework, e assim menos pressão é colocada para antecipar todas as demandas que possam ser colocadas em um determinado componente.

Ao reaproveitar experiências obtidas durante o desenvolvimento de projetos em um determinado domínio, podem ser percebidas semelhanças entre eles. Dessa forma, frameworks emergem a partir de tentativas de resolver problemas individuais, conforme surge a necessidade de resolver problemas relacionados, afirma Foote (1988). Por meio de experiências deste tipo, é possível discernir os fatores comuns presentes nas soluções de problemas específicos, e criar uma estrutura que incorpore essas semelhanças.

Corrêa (1997) define framework como uma arquitetura de software parcialmente acabada que forma uma infraestrutura de suporte para o desenvolvimento de aplicações de um domínio específico. Por sua vez, Coelho (2002) conceitua framework como um projeto reutilizável, que captura decisões comuns a um domínio de aplicação, podendo ser particularizado pelo desenvolvedor. De maneira geral, Johnson e Foote (1988) consideram um framework como um design abstrato, que se assemelha a um esqueleto e representa uma

solução para um tipo específico de problema, um modelo para um conjunto de problemas relacionados.

Foote (1988) compara que, enquanto uma biblioteca de classes contém um conjunto de componentes que podem ser reutilizados em contextos individuais, um framework contém um conjunto de componentes que devem ser reutilizados em conjunto para resolver um determinado tipo de problema. Frameworks permitem que projetos abstratos sejam personalizados e reutilizados, entretanto Fayad et.al. (2000) discorrem que um framework não se destina a ser usado como um aplicativo, em vez disso é a base para a construção deles.

Corrêa (1997) explica que aplicações baseadas em frameworks são construídas customizando-se algumas classes de forma que se antecipe uma parte do projeto de software. Ao contrário de bibliotecas de software convencionais, o controle da aplicação é feito em tempo de execução pelo framework e não pela aplicação. Programadores implementam o código chamado pelo framework em vez de invocarem diretamente bibliotecas de classe. De acordo com Johnson e Foote (1988), o framework muitas vezes faz o papel do programa principal ao coordenar e sequenciar as atividades da aplicação. Esta inversão de controle dá aos frameworks o poder de servir como esqueletos extensíveis. Entretanto, não se deve inferir que eles são úteis apenas para reutilização de código principal do aplicativo. A capacidade de permitir a extensão de componentes já existentes é, na verdade, uma de suas principais vantagens, afirma Foote (1988).

No entendimento de Fayad et.al. (2000), uma característica importante de um bom Framework é que ele fornece um ambiente de desenvolvimento maduro para o domínio em que será aplicado. Dessa forma, desenvolvedores de framework devem ter uma compreensão clara do domínio da aplicação em que o mesmo será empregado, e a elaboração e implementação devem refletir esse entendimento. Um framework adequado requer um código relativamente pequeno para atender aos requisitos do usuário para novos aplicativos.

Pelas características já apresentadas, a utlização de frameworks apresenta vários benefícios. Coelho (2002) ressalta que eles ajudam a padronizar a estrutura da aplicação e a reduzir o tempo de construção, o tamanho e complexidade do código-fonte. Corrêa (1997) acrescenta como vantagens a redução do tempo de codificação, depuração e teste de aplicações, e completa ainda dizendo que, ao ocultar complexidades fornecendo um nível alto de abstração, permite ao desenvolvedor concentrar-se na solução do problema. Por reduzirem a complexidade e o custo de sistemas, Fayad et al. (2000) afirmam que frameworks tornam-se ativos estratégicos para as organizações.

Entretanto, frameworks também apresentam desvantagens, entre elas Coelho (2002) destaca o alto custo de desenvolvimento comparado a uma aplicação específica. Outra desvantagem, segundo Corrêa (1997), é que eles reduzem a flexibilidade, uma vez que os componentes da aplicação devem seguir as restrições impostas pelo framework.

Não obstante, para que um framework desfrute de todas as vantagens apresentadas e seja verdadeiramente orientado a objetos, deve fornecer suporte a construções comuns, tais como polimorfismo, herança, encapsulamento e reutilização, afirmam Fayad et.al. (2000). Essas construções, quando bem aplicadas, maximizam a extensibilidade e customização. Ainda segundo os autores, a noção de extensibilidade é prescrita dentro do paradigma orientado a objetos e a customização é alcançada por meio de parametrização ou por pontos de extensão definidos na interface do framework.

Como frameworks emergem ao identificar as similaridades entre aplicações, é possível construir hierarquias de classe que reflitam essas semelhanças. Segundo Foote (1988), um framework pode ser estruturado como uma hierarquia de classes cada vez mais específicas, e é a capacidade que as hierarquias possuem de capturar relações à medida em que surgem que as tornam tão poderosas em ambientes que precisam enfrentar os requisitos voláteis.

Conforme afirma Foote (1988), ao permitirem que o código específico de um aplicativo resida em subclasses, Frameworks incentivam o desenvolvimento e posterior reforço das classes base. Frameworks são um dos melhores exemplos de como se deve explorar herança adequadamente numa linguagem orientada a objeto. Outras construções comuns dessas linguagens, quando bem aplicadas, também colaboram para o aparecimento de frameworks.

De acordo com Johnson e Foote (1988), cada hierarquia de classes oferece a possibilidade de tornar-se um framework. O comportamento de um aplicativo específico é definido pela adição de subclasses à hierarquia, e pela implementação dos métodos contendo as especificidades da aplicação. Cada método adicionado a uma subclasse deve respeitar as convenções internas de suas superclasses. Esse tipo de framework é chamado de caixa-branca, porque a sua implementação deve ser entendida para usá-los, é necessário compreender a estrutura interna das classes que precisam ser estendidas.

Um framework caixa-branca é apenas um conjunto de convenções para substituição de métodos, não existe uma linha tênue entre um framework caixa-branca e uma hierarquia de classes simples, ressaltam Johnson e Foote (1988). Na sua forma mais simples,

um framework caixa-branca é o esqueleto de um programa, e as subclasses são as adições ao esqueleto. Os autores expõem que o principal problema relacionado a esse tipo de framework é que eles costumam apresentar uma curva de aprendizado mais acentuada, uma vez que aprender a usá-lo é o mesmo que aprender como ele é construído.

Por outro lado, se o relacionamento entre as partes de um framework for definido em termos de um protocolo abstrato - ou assinatura - em vez de usar herança, a generalidade entre as partes do framework é reforçada. Foote (1988) afirma que conforme a compreensão acerca de um sistema aumenta, relações baseadas em componentes devem surgir para substituir aquelas baseadas em herança. Dessa forma, frameworks também podem ser compostos por um conjunto de componentes, que interagem para fornecer o comportamento específico do aplicativo. Frameworks baseados em componentes são conhecidos como caixa-preta.

Neste tipo de framework, cada um dos componentes deve respeitar um protocolo particular. As interfaces entre os componentes devem ser definidas por um protocolo, para que o usuário precise entender apenas a interface externa dos mesmos (FOOTE, 1988). Contudo, Johnson e Foote (1988) destacam que embora os frameworks caixa-preta sejam mais fáceis de aprender a usar, eles são menos flexíveis.

Coelho (2002) sugere que a utilização de componentes pode diminuir a complexidade da arquitetura de um framework. Ao definir interfaces entre os componentes, o acoplamento entre eles é reduzido, o que, segundo Foote (1988), aumenta a coesão e generalidade do framework, pois permite que sua estrutura funcione com qualquer objeto compatível. Conforme afirmam Johnson e Foote (1988) um framework se torna mais reutilizável quando a relação entre suas partes é definida em termos de um protocolo, em vez de herança.

Para Foote (1988), um framework pode, em casos ideais, evoluir a partir do tipo de estrutura de caixa-branca para a de caixa-preta. De acordo com o autor, frameworks caixa-branca devem ser vistos como uma etapa natural na evolução de um sistema, e é importante perceber que, proporcionando um meio-termo entre as características comuns e as específicas de um projeto, frameworks caixa-branca fornecem um caminho indispensável ao longo do qual podem evoluir.

2.3.1 Frameworks e Sistemas Empresariais

Em revisão feita por Fayad et al., (2000) acerca de frameworks para desenvolvimento de aplicações empresariais, foi revelado que muitos deles possuem um catálogo de objetos prontos para usar, que podem ser usados para alterar o comportamento do sistema. Estes objetos também devem estar disponíveis como modelos para a construção de novos objetos de negócios com base em cenários capturados nos requisitos funcionais. Os autores apresentam o conceito de tema de negócio duradouro (Enduring Business Theme – EBT), que é a estrutura para as sentenças que são encontradas na línguagem dos negócios. Os objetos de negócio são instâncias de temas de negócio, que preenchem atributos ou semântica dessas sentenças. O tema de negócios está atrelado a uma operação de negócios do ponto de vista do cliente, que se preocupa mais com o que está sendo realizado, e só em menor medida como está sendo feito.

Bussiness Objects (BO) são instanciações detalhando como os negócios são realizados em um ponto discreto no tempo. Assim, um bom framework empresarial reflete a compreensão concisa das qualidades duradouras do negócio por meio de EBTs mais implementações discretas que reflitam as melhores práticas conhecidas (FAYAD et al., 2000).

Um tema de negócios verdadeiramente duradouro é aquele que sobrevive ao longo do tempo. Para Fayad et al., (2000), o critério mais importante para a identificação de EBTs e BOs é que ambos devem ser estáveis ao longo do tempo, diferindo na forma como são estáveis. Temas de negócios são completamente estáveis tanto interna como externamente, dificilmente mudam. Os objetos de negócios podem mudar internamente. Externamente, no entanto, devem permanecer os mesmos. Alterar esses objetos não deve propagar alterações por todo sistema. Os comportamentos são adaptados às necessidades por meio de diferentes objetos de negócio, cada um representando um papel no contexto de cada cliente.

2.4 Arquitetura de Software

Conforme o tamanho dos sistemas de software aumenta, o problema de projetar extrapola os algoritmos e estruturas de dados. Assim, a concepção e especificação da estrutura geral do sistema emergem como um novo tipo de problema. Questões estruturais incluem a organização e a estrutura de controle global, protocolos de comunicação, sincronização e acesso a dados; atribuição de funcionalidade a elementos de design, distribuição física, composição de elementos de design, escalabilidade e desempenho, e

seleção entre as alternativas de design. Esse é o nível de projeto de arquitetura de software (GARLAN; SHAW, 1994).

Pode-se definir a arquitetura como a primeira atividade a ser realizada na concepção de um projeto de software. Trata-se do momento em que são tomadas as primeiras decisões sobre a forma pela qual os requisitos funcionais e não funcionais deverão ser atendidos pelo produto final (BARCELOS, apud MÜLLER, 2006). Para Garlan (2000), uma boa arquitetura pode ajudar a assegurar que o sistema irá satisfazer requisitos chave em áreas como performance, confiabilidade, portabilidade, escalabilidade e interoperabilidade. Garland e Anthony (2003) acrescentam outros atributos ou qualidades de interesse na arquitetura, dentre os quais se destacam segurança, facilidade de manutenção e usabilidade.

Ainda conforme o pensamento de Garlan (2000), existem muitas definições de arquitetura de software, mas no núcleo de todas elas está a noção de que a arquitetura de um sistema descreve sua estrutura bruta. Adicionalmente, uma descrição arquitetural inclui informação suficiente para permitir análise de alto nível e avaliação crítica. Ao fornecer uma descrição abstrata de um sistema, a arquitetura expõe determinadas propriedades, enquanto omite outras. Idealmente, esta representação fornece um guia tratável para o sistema como um todo, permite raciocinar sobre a capacidade do sistema para satisfazer certas exigências, e sugere um plano para sua construção.

De acordo com Garlan (2000), uma boa arquitetura é um fator crítico de sucesso para a concepção e desenvolvimento de um sistema. Segundo ele, a arquitetura assume um papel importante em pelo menos seis aspectos do desenvolvimento de software:

a) entendimento: amplifica a capacidade de compreender grandes sistemas apresentando-os sob um nível de abstração em que o sistema pode ser facilmente entendido. Além disso, no seu ápice, a descrição arquitetural expõe as restrições de alto nível no projeto do sistema, bem como razões para fazer escolhas arquiteturais específicas;

b) reuso: o projeto arquitetural suporta reutilização de grandes componentes e também Frameworks em que componentes podem ser integrados;

c) construção: uma descrição arquitetural fornece um plano parcial para o desenvolvimento indicando os principais componentes e as dependências entre eles. Documenta limites de abstração entre partes do sistema, identificando as principais interfaces internas do sistema e quais partes podem depender de serviços providos por outras partes;

d) evolução: a arquitetura pode expor as dimensões em que se espera que o sistema evolua. Explicitando os limites de um sistema, os desenvolvedores podem ter um

melhor entendimento das ramificações das mudanças, e estimar custos de modificações mais precisamente;

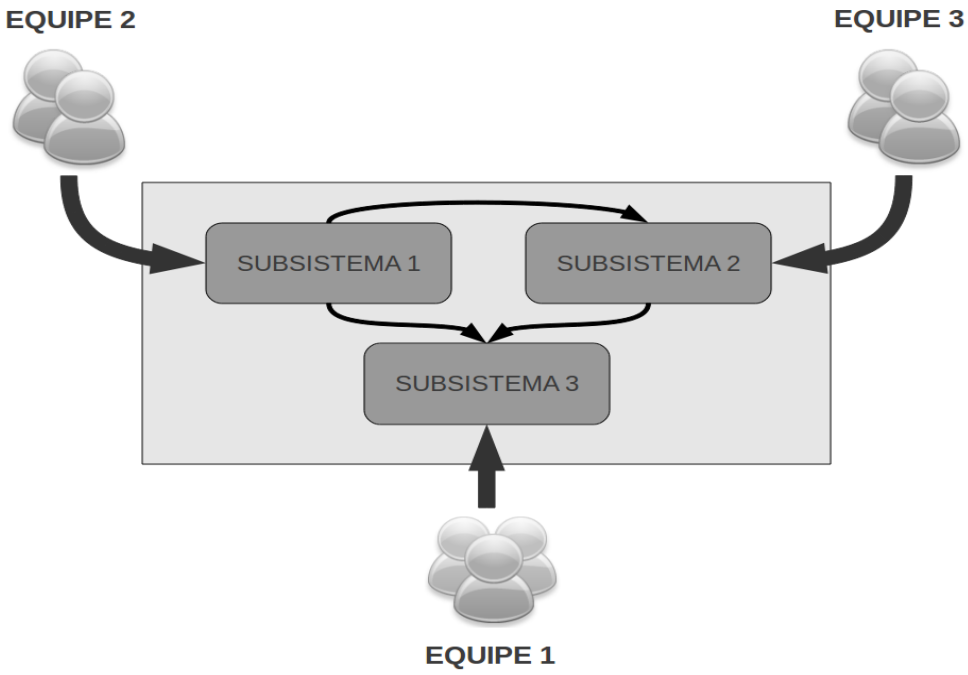
e) análise: descrições arquiteturais fornecem novas oportunidades para análise, incluindo checagem de consistência, conformidade com restrições impostas ao estilo arquitetural, conformidade a atributos de qualidade, análise de dependência, e análise específicas de domínio para arquiteturas construídas em estilos específicos;

f) gerenciamento: avaliação crítica de uma arquitetura tipicamente leva a um entendimento mais claro dos requisitos, estratégias de implementação e riscos potenciais.

O design arquitetural do sistema preocupa-se com descrever sua decomposição em elementos computacionais e suas interações. Tarefas de projeto nesse nível incluem organizar o sistema como uma composição de componentes; desenvolver estruturas globais de controle; escolher protocolos para comunicação; sincronização e acesso a dados; distribuição física de componentes; escalar o sistema e estimar performance; e selecionar entre alternativas de design (GARLAN et al., 1995).

Na opinião de Malan e Bredemeyer (2010), a arquitetura do software deve tornar o sistema mais compreensível e intelectualmente gerenciável – fornecendo abstrações que omitam detalhes desnecessários, unificando e simplificando conceitos, decompondo o sistema. Devem ser considerados fatores como o número de pessoas envolvidas em todos os aspectos da construção do sistema, dependências no projeto, uso de terceirização, equipes geograficamente distribuídas etc. A figura 1 apresenta uma visão geral das responsabilidades das equipes, permitindo visualizar necessidades de comunicação ao estabelecer as dependências entre os subsistemas.

Figura 1 – Visão geral de um software modular



Fonte: Desenvolvimento próprio

Conforme pode ser visto na figura 1, a arquitetura deve permitir o gerenciamento de um software de um nível mais alto, focando nos subsistemas e suas dependências. Outra maneira de pensar a arquitetura de software é partir de questões típicas que podem ser respondidas por visões da arquitetura, tais como as levantadas por Garland e Anthony (2003): Quais são os componentes, suas responsabilidades e as interfaces por eles usadas e fornecidas? Ao realizar uma mudança, quais componentes e equipes serão impactados?

No coração da arquitetura de software está o princípio da abstração: esconder alguns detalhes do sistema por meio do encapsulamento a fim de identificar e sustentar suas propriedades. Um sistema complexo conterá muitos níveis de abstração, cada um com sua própria arquitetura. Uma arquitetura representa uma abstração do comportamento do sistema naquele nível, tal que elementos arquiteturais são delineados por interfaces abstratas fornecidas a outros elementos desse nível (FIELDING, 2000).

Fielding (2000) acrescenta que a complexidade dos softwares modernos tem colocado em foco sistemas componentizados, em que a implementação é particionada em componentes independentes que se comunicam para realizar uma tarefa. Pesquisas em arquitetura de software investigam métodos para determinar como particionar melhor um

sistema, como componentes identificam e se comunicam com outros, como a informação é comunicada e como elementos de um sistema podem evoluir independentemente.

Neste âmbito, a modularidade desempenha um papel importante no desenvolvimento de sistemas, principalmente em relação à evolução do software. Muitos problemas que surgem no desenvolvimento de software são decorrentes do alto acoplamento resultante do baixo grau de modularidade do sistema, causando atrasos na implantação e aumento dos custos. Assim, é importante aprofundar esta discussão, tendo em vista a necessidade de fornecer uma visão de alto nível do sistema. A seguir, será detalhada a importância da modularidade como possibilidade de gerenciamento de um software.

2.4.1 Modularidade como um Facilitador da Customização

Modularidade é uma propriedade importante em todo tipo de software, principalmente em sistemas complexos e de grande porte, pois permite que sejam decompostos em partes menores e mais gerenciáveis. Talevski et al. (2004) citam que mais de 50% dos projetos de software de grande porte costumam falhar, pois à medida que aumenta o tamanho de um software, aumentam também sua complexidade e a dificuldade em mantê-lo. Dessa forma, para gerenciar a complexidade de um grande sistema, é desejável particionar os vários subsistemas em módulos independentes ou semi-independentes, e fazer com que se comuniquem de alguma maneira.

Neste contexto, módulos ou subsistemas são conjuntos de funcionalidades relacionadas, que oferecem serviços para outros módulos. Garland e Anthony (2003) apontam que módulos fornecem serviços por interfaces bem definidas, encapsulam tanto estado quanto comportamento e dependem apenas de um Framework ou do sistema operacional para fornecer comunicação a outros componentes. Além disso, frequementente possuem muitas opções de configuração que impactam o seu comportamento e são desenvolvidos, testados e distribuídos independentemente.

Módulos mostram-se importantes para a evolução de um sistema, pois, de acordo com Garland e Anthony (2003), o módulo é a unidade física de substituição. Na opinião de Zhu (2005), a utilização de módulos é uma abordagem promissora para a customização de software, permitindo a construção de aplicativos através de módulos acopláveis (plug and play); desta forma, é possível produzir uma extensa variedade de produtos, pela combinação de um número limitado de módulos. Ademais, Talevski et. al. (2004) referem que a engenharia de software baseada em módulos é uma maneira de elevar o

nível de abstração, de modo que este possa ser construído a partir de componentes já existentes.

Vários autores reforçam a importância da modularidade no desenvolvimento de software. Johnson e Foote (1988) acreditam que a modularidade faz com que seja mais fácil entender o efeito das mudanças em um programa. Tulach (2008), por sua vez, afirma que dividir uma aplicação em módulos pode tornar o projeto mais claro, melhorar o controle das dependências, e permitir maior flexibilidade durante a manutenção. Além disso, Garland e Anthony (2003) enfatizam que subsistemas são importantes porque permitem acelerar o desenvolvimento, pois possibilita que equipes trabalhem em paralelo. Martin e Martin (2006) pontificam que o projeto de grandes sistemas depende criticamente do bom design de módulos, de forma que as equipes possam focar em módulos isolados em vez de se preocupar com todo o sistema.

Sob este viés, um sistema modular é, portanto, representado como um complexo de subsistemas, nos quais se procura minimizar e padronizar as interdependências entre eles. Benkler (2002) define modularidade como a capacidade que um software possui de ser dividido em unidades menores de tal maneira que possa ser desenvolvido de forma independente e assíncrona.

Espera-se ainda que a modularidade traga outros benefícios, e Parnas (1972) ressalta os seguintes:

a) O tempo de gerenciamento deve ser reduzido, pois equipes poderão trabalhar separadamente em cada módulo, com pouca necessidade de comunicação. Consequentemente, o custo global do software é reduzido;
b) O produto torna-se mais flexível, permitindo fazer mudanças drásticas em um módulo sem a necessidade de mudar outros;
c) A compreensão acerca do sistema é facilitada, pois cada módulo pode ser estudado separadamente, fornecendo uma visão de alto nível e permitindo reconhecer rapidamente a estrutura do software.

No entanto, Garland e Anthony (2003) asseveram que para experimentar estas características é desejável ter subsistemas com alta coesão e baixo acoplamento. Ao particionar um sistema, os elementos devem ser agrupados para maximizar os aspectos coesivos e minimizar o acoplamento dos elementos, principamente se os subsistemas forem desenvolvidos por equipes separadas. Se o acoplamento for inapropriado, as equipes gastarão tempo e esforços desnecessários negociando interfaces entre os subsistemas.

Para que os módulos possam ser compreendidos isoladamente, é necessário que eles sejam simples, e suas responsabilidades estejam bem definidas. Desta necessidade, emerge a interface do módulo, que expõe as responsabilidades na forma de serviços. Ela permite que processamento ocorra de forma transparente, sem que outros módulos necessitem saber “como”; tudo o que necessitam saber é “o que” é feito, sem precisar interagir com a implementação interna (FIELDING, 2000).

A interface pode ser vista como um contrato entre módulos. Conforme Talevski et al (2004), além de especificar os serviços oferecidos por um módulo, uma interface indica o protocolo de interação, e as condições requeridas que devem ser garantidas para que os módulos interoperem. Dessa maneira, desde que se continue a aderir ao contrato de interface, é possível substituir ou modificar a implementação interna sem afetar o comportamento de outros módulos.

Parnas (1972) reforça que para alcançar esse objetivo, a interface deve revelar o mínimo possível sobre seu funcionamento interno. O autor sugere que sistemas sejam decompostos de forma que uma decisão de design seja ocultada. A isso ele denomina “ocultação de informação”, que recomenda tratar módulos de software como entidades opacas, cuja informação relevante está disponível apenas internamente, não sendo acessível a componentes externos. Narduzzo e Rossi (2003) pontificam que todas as informações sobre o design e funcionamento do módulo não são comunicadas.

Um grande problema é decidir o que será exposto pela interface. Parnas et.al. (1985) afirmam que detalhes que são suscetíveis à mudança devem ser os “segredos” do módulo; os únicos pressupostos que devem aparecer nas interfaces são aqueles com menor probabilidade de mudança. Assim, deve ser possível fazer alterações previstas sem modificar as interfaces do módulo; mudanças menos prováveis podem implicar modificações na interface, mas apenas para os módulos que são pequenos e não utilizados amplamente. Apenas as alterações muito improváveis devem exigir mudanças nas interfaces de módulos muito utilizados. Os autores finalizam afirmando que, exceto para mudanças na interface, os programadores não devem precisar se comunicar ao trabalhar em módulos individuais, colaborando para o desenvolvimento assíncrono.

No entanto, segundo Parnas et.al. (1985), aplicar o princípio de ocultação de informação nem sempre é fácil, pois requer que a possibilidade de mudanças seja estimada. Essas estimativas são baseadas em experiências passadas, e podem exigir um conhecimento profundo da área da aplicação. Na concepção de Garland e Anthony (2003), um dos aspectos mais críticos durante o desenvolvimento da arquitetura é entender o que está propenso a

mudanças e o que não mudará. Um dos perigos da mudança é que ela requer suposições sobre eventos futuros. Adicionar flexibilidade à arquitetura no início do projeto custará tempo e dinheiro, e deve ser considerado o custo de fazer as mudanças depois, refatorando o projeto. Na opinião de Barstow e Arango (1991), o investimento para adicionar esta flexibilidade pode ser alto, mas é justificado quando for antecipada a necessidade de um grande número de customizações.

Compreendendo o papel da modularidade no desenvolvimento de grandes sistemas, busca-se estudar maneiras de tirar proveito de suas características para facilitar a customização de sistemas. Barstow e Arango (1991) referem que o ideal é que customizações envolvam simplesmente adicões e extensões em vez de modificações, de forma que as mesmas adições e extensões feitas sejam diretamente incorporadas às customizações. Uma afirmação importante é que o código customizado deve ser mantido separadamente do código original do programa. Estudo realizado por Dittrich e Vancouleur (2008) aponta que é uma boa prática separar o código customizado do código padrão do sistema; isso significa que uma nova funcionalidade é implementada em classes ou objetos adicionais. Dessa forma, o código customizado pode ser armazenado em um módulo independente e ser adicionado dinamicamente e em tempo de execução. Esse objetivo pode ser atingido por meio das técnicas relatadas a seguir.

2.4.2 The Open/Closed Principle (OCP) e o Padrão Plugin

A abstração desempenha um papel fundamental no que diz respeito ao desenvolvimento orientado a objetos. Ela permite que certos detalhes sejam omitidos de outros módulos, colaborando com o encapsulamento que também é vital para a manutenção do sistema. Para Martin e Martin (2006), a maneira mais comum de estender o comportamento de um módulo é fazer alterações em seu código, porém é possível alterar o comportamento por meio de abstrações. Em linguagens orientadas a objetos é possível criar abstrações que representam um grupo ilimitado de comportamentos possíveis.

Meyer (1997) formalizou esta característica de fazer alterações no comportamento como o princípio aberto/fechado (The Open/Closed Principle - OCP), em que entidades de software (classes, módulos, funções) devem estar abertas para extensão, mas fechadas para modificação. Estar aberto para extensão implica dizer que, conforme os requisitos da aplicação mudam, pode-se enriquecer o módulo com novos comportamentos que

satisfazem essas mudanças. Estar fechado para modificação significa mudar o que o módulo faz sem alterar códigos existentes.

Martin e Martin (2006) consideram que o OCP é o coração do design orientado a objetos. Os mecanismos primários por trás do OCP são abstração e polimorfismo obtidos por meio da herança, tornando possível criar classes derivadas que implementam métodos abstratos nas classes base. Assim, é possível estabelecer uma separação clara de funcionalidade genérica e implementação detalhada. A conformidade a esse princípio é o que rende os maiores benefícios disseminados pela tecnologia orientada a objetos: flexibilidade, reuso, capacidade de manutenção e robustez.

Conforme Johnson e Foote (1988), a especificação de um objeto é dada pelo seu protocolo, ou seja, o conjunto de mensagens que podem ser enviadas a ele. Objetos com o mesmo protocolo são intercambiáveis, ou seja, podem ser trocados em tempo de execução. Protocolos muitas vezes são representados por classes abstratas, que são classes que não possuem instâncias. Classes abstratas definem os métodos que devem ser implementados por suas subclasses, e podem ser usadas como esqueletos.

Foote (1988) menciona que a herança permite que extensões sejam feitas deixando o código original intacto. A especialização e capacidade de reutilização fornecidas pela herança e polimorfismo aumentam o potencial de aplicação dos componentes pré-existentes. Acrescenta ainda que variantes globais podem ser eliminadas usando uma hierarquia. No entendimento de Martin e Martin (2006), adicionar novas funcionalidades representa uma tarefa simples, pois ao criar uma nova classe derivada, não há necessidade de procurar em toda a aplicação por lugares que precisam ser alterados. Nenhum módulo existente precisa ser modificado.

Segundo Fayad et.al. (2000), a maior parte da engenharia realizada durante um projeto de software deve ser focada nas áreas que permanecerão estáveis. Esta abordagem garante um núcleo estável e, portanto, um produto de software estável. Essas alterações que são introduzidas no projeto de software ocorrerão na periferia, uma vez que o núcleo foi baseado em algo que permanece estável. Dessa forma, evita-se o grande esforço de reengenharia para pequenas alterações.

Na opinião de Martin e Martin (2006), de forma geral, não importa o quanto um módulo seja fechado, sempre haverá algum tipo de mudança para a qual ele não está preparado. Como o fechamento não pode ser completo, deve ser estratégico. Nesta perspectiva, deve-se escolher os tipos de mudanças que serão cobertas entre os tipos mais prováveis, e construir abstrações para proteger-se delas. É preciso conhecer bem os usuários e

o segmento para julgar a probabilidade dos vários tipos de mudanças, visto que a conformidade ao OCP é custosa, toma tempo e esforço para criar abstrações apropriadas. As abstrações também aumentam a complexidade do design, pois há um limite do número de abstrações que os desenvolvedores podem permitir. Resistir à abstração prematura é tão importante quanto a abstração propriamente dita. Por estas razões, é preciso limitar a aplicação do OCP a mudanças mais prováveis. Para Barstow e Arango (1991), construir flexibilidade para acomodar mudanças arbitrárias é muito caro e pode ser um desperdício. A melhor estratégia é evitar generalizações desnecessárias e fazer com que o sistema seja flexível apenas nas áreas mais propícias à mudança.

A opção pelo OCP implica a reorganização do sistema para que outras mudanças similares não causem mais modificações no futuro. Se o princípio for bem aplicado, alterações subsequentes serão alcançadas adicionando código novo, sem necessidade de alteração do código em funcionamento. Existem estratégias relativamente simples e efetivas para alcançar esse ideal. Os padrões TEMPLATE METHOD e STRATEGY, descritos por Gamma et al., (1994), são as formas mais comuns de satisfazer o OCP. Ambos os padrões resolvem o problema de separar um algoritmo genérico de um contexto detalhado, que é uma necessidade frequente em projetos de software. O algoritmo genérico e a implementação detalhada devem depender de abstrações (MARTIN; MARTIN, 2006).

O padrão PLUGIN descrito por Fowler et al., (2002) tem como propósito fornecer a ligação entre classes em tempo de execução por meio de configurações. Para isto, faz-se necessário definir com interfaces separadas comportamentos que demandam implementações diferentes. Utiliza o padrão FACTORY, o qual exige que as instruções de ligação estejam contidas em um ponto externo único, para que a reconfiguração não implique em uma reconstrução. Um arquivo de texto pode suprir essa necessidade, especificando uma implementação da interface requisitada e permitindo que a implementação desejada seja obtida. Segundo Fowler (2004), quando houver necessidade de adiar a escolha da classe de implementação, é possível utilizar o padrão PLUGIN. É essencial que a montagem seja feita separadamente do resto da aplicação para que seja possível escolher entre diferentes comportamentos.

O padrão PLUGIN vale-se do princípio da injeção de dependências, cujo objetivo, de acordo com Cruz (2007), é remover o controle de dependências de dentro das classes. Ao invés de instanciar cada objeto internamente, simplesmente recebe os objetos necessários como parâmetro de construtor, ou por meio de métodos de acesso. Desta forma, não é necessário instanciar todas as suas dependências, apenas descrever como elas devem ser

criadas. Não há necessidade de acoplar os componentes e serviços diretamente no código, basta descrever que serviços são requeridos por cada componente. Fowler (2004) afirma que o principal benefício da injeção de dependências é remover a dependência que uma classe apresenta em relação a uma implementação concreta. O uso de injeção de dependências pode facilitar a visualização das dependências de um componente, bastando olhar para o mecanismo de injeção.

A vantagem de usar injeção de dependências é que a instanciação dos componentes é separada. É uma ferramenta eficaz de configuração que permite escolher o comportamento em tempo de execução, adiando a escolha da implementação. Permite especificar em um nível genérico como um grupo de objetos é interligado. Entretanto, esta capacidade de adiar a decisão de qual código concreto será executado só é alcançada em linguagens orientadas a objeto por meio de abstrações. A utilização de injeção de dependências encoraja o desenvolvimento de interfaces em vez de implementações, resultando em código menos acoplado, que é mais fácil de modificar e testar. O ganho é a habilidade de injetar dependências em qualquer objeto sem fazer alterações no código que as suporta.

2.5 Reflexão Computacional e Geração de Código

Como visto, concepção e desenvolvimento de software que pode se adaptar a mudanças é uma tarefa difícil. Se for possível utilizar algum artifício para possibilitar que objetos dentro de um sistema sejam autodescritos e permitir que esta descrição seja modificada dinamicamente, obtém-se uma ferramenta poderosa para lidar com customizações. Uma forma de alcançar estas características é a reflexão computacional, definida por Corrêa (1997) como a habilidade de um sistema de representar, operar e tratar sua própria computação da mesma forma que ele representa, opera e trata sua aplicação, permitindo estender ou modificar a semântica para que ela se ajuste a necessidades específicas. Assim reflexão permite que um sistema seja definido de uma forma que se possa lidar automaticamente com situações imprevistas.

Segundo Corrêa (1997), reflexão fornece um mecanismo que permite a um sistema manipular ou modificar seu comportamento devido à incorporação de estruturas representado seu próprio estado. Estas estruturas são conhecidas como metadados e podem fornecer informações sobre os atributos, métodos, e outras características sobre os objetos no sistema; desta forma, abre caminho para que a estrutura e o comportamento do sistema sejam

alterados dinamicamente ao manipular estes metadados. De acordo com Maes (1987), em uma arquitetura reflexiva, todos os aspectos estruturais e comportamentais do sistema são armazenados em metaobjetos e separados da lógica da aplicação.

Desta forma, uma arquitetura reflexiva é organizada em camadas: o metanível contém os metaobjetos que encapsulam a estrutura e comportamento variável e o nível de base contém os componentes de lógica da aplicação que dependem dos metaobjetos. Foote (1994) explica que objetos do metanível, ou metaobjetos, são objetos que definem, implementam ou apoiam a execução da aplicação. Conforme Corrêa (1997), esta arquitetura introduz um novo estilo de programação em que metaobjetos armazenam informações sobre objetos do nível-base. O nível base ou nível de objeto, associado ao domínio da aplicação, é ideal para implementar requisitos funcionais; e metanível está relacionado com a solução de problemas e armazenamento de informação sobre o nível base.

Segundo Corrêa (1997), para cada objeto pode existir um metaobjeto que representa seus aspectos estruturais e comportamentais. Cada mensagem enviada ao objeto é interceptada e dirigida a seu metaobjeto, que se encarrega de sua execução. O emissor da mensagem não toma conhecimento da computação reflexiva: ele envia a mensagem solicitando os serviços de um objeto e recebe o resultado esperado, sem saber que a mensagem foi desviada pelo metaobjeto. Na opinião do mesmo autor, essa noção é um modo interessante e conveniente de implementar de forma transparente requisitos não funcionais (administrativos), como tolerância a falhas, depuração de erros, otimização e políticas de segurança. Requisitos administrativos podem ser mudados sem interferir nos requisitos funcionais. De acordo com Maes (1987), a transparência é outra característica fundamental de todos os sistemas reflexivos, e refere-se ao fato de que as entidades em cada nível são completamente alheias à presença e funcionamento de entidades de níveis mais elevados.

Assim, segundo Corrêa (1997), arquiteturas reflexivas possibilitam a simplificação dos protocolos presentes no Framework, separando-se as definições referentes às classes das definições das interdependências entre elas. Dessa forma o comportamento dos objetos condicionados a uma única classe é descrito normalmente pelas classes base, mas o comportamento dos objetos condicionados a várias classes é descrito pelas classes no metanivel.

Foote (1994) pondera que uma linguagem com suporte à reflexão permite que um programa em execução inspecione e mude os objetos dos quais ele é construído. A introspecção é a capacidade de avaliar, mas não alterar, os objetos que implementam um sistema. Um programa reflexivo orientado a objetos pode acessar os objetos que definem seu

funcionamento, e alterá-los dinamicamente, se necessário. Uma linguagem que suporta a redefinição dinâmica de partes existentes do sistema é dita mutável. Uma linguagem que suporta a adição de novas funcionalidades, mas não permite a redefinição das já existentes é dita extensível.

Corrêa (1997) destaca a existência de uma sobrecarga associada aos mecanismos de reflexão, que podem impactar o desempenho da aplicação. Garland e Anthony (2003) discutem que existem outros custos relacionados, como dificuldade de rastrear dependências e complexidade da depuração, pois erros que poderiam ser encontrados em tempo de compilação, em um sistema baseado na reflexão devem ser encontrados em tempo de execução.

Aliada à reflexão computacional, a geração de código mostra-se importante para a customização de sistemas. Na opinião de Zhu (2005), geração automática de código é essencial para a customização de software. Normalmente, é realizada por um compilador chamado de gerador de código, que traduz as especificações de uma aplicação em código-fonte e funciona como um mapeamento entre o modelo de negócio e uma linguagem de programação.

Entre as várias técnicas que utilizam geração de código, uma das mais promissoras é o Desenvolvimento Guiado pelo Modelo (Model Driven Development – MDD). Segundo Cicchetti et al. (2007), é uma abordagem de desenvolvimento em que as funcionalidades são definidas a partir de modelos independentes de plataforma, traduzidos para modelos específicos de uma plataforma que podem ser executados pelo computador.

Conforme Cicchetti et al. (2007), o objetivo do MDD é produzir código a partir de modelos abstratos; entretanto, os autores salientam que as ferramentas atuais nem sempre são capazes de criar automaticamente aplicativos completos, exceto para casos muito específicos. Uma das principais razões é que o poder expressivo de modelos estruturais não é suficiente para suportar a complexidade dos requisitos que precisam ser expressos. Apesar das limitações, as ferramentas MDD permitem gerar o código de infraestrutura, bem como uma aplicação completa de trabalho que suporta operações básicas sobre uma entidade, como criação, leitura, atualização e exclusão (CRUD - Create, Retrieve, Update, Delete), e algumas operações de comportamento. A utilização de modelos abstratos e conjunto com reflexão permite que códigos sejam gerados dinamicamente em tempo de execução, possibilitando a alteração do próprio sistema.

2.6 CONSIDERAÇÕES

Um grande desafio na área de engenharia de software é lidar com mudanças nos requisitos. Espera-se que softwares sejam construídos rapidamente, com baixo custo e alto nível de qualidade. Buscar soluções que atendam todas as necessidades do cliente pode não ser uma tarefa simples. Além disso, oferecer personalizações de um sistema a vários clientes de áreas de negócio distintas tem se mostrado um desafio. Como pôde ser visto, a modularidade desempenha um papel importante em se tratando de customização de software. Aliada a outras técnicas consolidadas na engenharia de software como reflexão computacional e geração de código, obtêm-se um substracto consistente para facilitar a customização e expansão de um software.

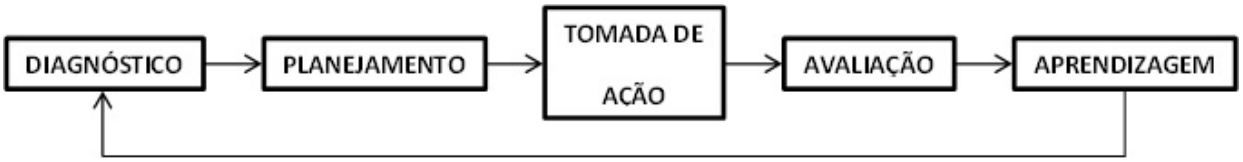
3 PROPOSTA DE UM MECANISMO DE APOIO À CUSTOMIZAÇÃO DE SISTEMAS WEB

Tendo em vista a necessidade de facilitar a elaboração e manutenção de software para atender às diferentes necessidades dos usuários, neste capítulo é apresentada uma proposta de um mecanismo para apoiar a customização de sistemas web, a partir da combinação de outras técnicas conhecidas na engenharia de software. A partir das necessidades observadas em relação à customização na empresa estudada, a proposta foi elaborada de forma que funcionalidades pudessem ser adicionadas e removidas com pouco esforço. No mecanismo proposto, a customização é alcançada por meio da combinação de modelos, mantendo o código customizado separado do original. Para o desenvolvimento da proposta, foi utilizada a metodologia pesquisa-ação (SUSMAN; EVERED, 1978), cujas etapas são delineadas na sequência.

3.1 METODOLOGIA PESQUISA-AÇÃO

A estrutura básica que guia a metodologia pesquisa-ação é descrita num processo de cinco fases, apresentado na figura 2. Conforme Travassos e Medeiros (2010), a metodologia permite sincronizar pesquisa e intervenções organizacionais, isto é, ela oferece um cenário de pesquisa mais experimental, envolvendo um contexto real da empresa para investigar resultados de ações concretas.

Figura 2 – Fases da metodologia pesquisa-ação



Fonte: Travassos e Medeiros (2010)

Em primeiro plano, o diagnóstico é a definição do campo e do tema de pesquisa, dos interessados e suas expectativas e da área do conhecimento a ser abordado. A seguir, é efetuado o planejamento, onde são definidas as ações a serem tomadas. Logo após, são executadas as ações planejadas na tomada de ação, e é feita a avaliação de seus efeitos

frente ao apoio teórico utilizado na definição das ações. Finalmente, a aprendizagem envolve a partilha do conhecimento adquirido por toda a organização (TRAVASSOS; MEDEIROS, 2010).

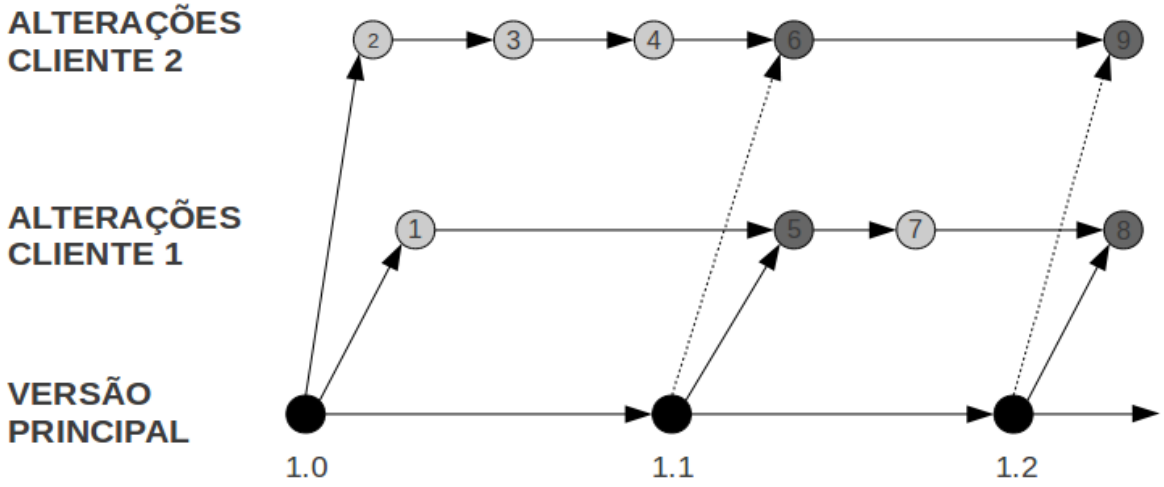
3.1.1 Diagnóstico

Em levantamento efetuado nos registros de desenvolvimento da empresa, foi constatado que os tipos de customização mais comuns são adicionar campos a cadastros, desenvolver novos cadastros e alterar relatórios. Interfaces gráficas são um dos aspectos que mais mudam, pois são a parte visível ao usuário. Um dos principais casos de customização encontrados está relacionado à alteração dos formulários, essenciais para interação com o sistema, pois permitem que dados sejam enviados facilmente para a aplicação. Muitas vezes é necessário capturar informações adicionais; assim, é comum que sejam adicionados ou removidos campos destes elementos. Outro caso comum é a alteração de relatórios, visando fornecer uma visão diferenciada dos dados, que seja mais adequada à tomada de decisões. A análise revelou ainda que aproximadamente 70% do total das funcionalidades são cadastros do tipo CRUD. Pelo fato de as empresas em que o software foi implantado atuarem em áreas parecidas (indústria de transformação) e de tamanho similar, não foram observadas customizações agressivas nos processos do sistema, pois este se mostrou adequado à realidade das empresas.

Os principais problemas encontrados durante o desenvolvimento estão relacionados a falhas na gerência de configuração e versionamento do projeto. O primeiro está relacionado ao alto acoplamento entre as funcionalidades, decorrente da falta de definição das interfaces entre módulos, o que dificulta a separação de funcionalidades. Como parte da estratégia comercial, a empresa pretende vender módulos separadamente, para atender necessidades específicas, caso o cliente necessite de apenas um subconjunto do software. Atualmente, o sistema é desenvolvido de forma monolítica, não existindo uma separação efetiva dos módulos, a separação é apenas conceitual.

Um segundo problema diz respeito à maneira adotada para lidar com a necessidade de customizações, que se provou custosa e propensa a erros, além de dificultar o versionamento do software. As customizações são mantidas paralelamente ao código principal na forma de ramificações (branches) no sistema de controle de versões utilizado, o GIT (Global Information Tracker). A figura 3 ilustra tal situação.

Figura 3 – Alterações mantidas em branches separados



Fonte: Desenvolvimento próprio

Em primeiro plano, é apresentada a primeira versão estável do ERP, a 1.0. Ao surgir a necessidade de modificar alguma funcionalidade para o Cliente 1, é criada uma ramificação (branch) no sistema de controle de versões, e as alterações são efetuadas, divergindo da versão principal (1). Paralelamente, foram necessárias três alterações para adequar o sistema ao Cliente 2, o que gerou mais uma ramificação com o código customizado (itens (2), (3) e (4)). No entando, a versão principal continua a evoluir, defeitos são corrigidos e novas funcionalidades são adicionadas na versão 1.1; estas evoluções precisam ser reincorporadas às versões customizadas para cada cliente. Isso é feito por meio da operação de mesclar (merge) no GIT, representado pelos itens (5) e (6).

Embora o GIT seja eficiente e facilite ao efetuar essa operação, ainda há necessidade de intervenção por parte do programador, pois devem ser levadas em consideração as customizações que foram feitas para cada cliente. Não é difícil perceber que conforme o número de clientes aumenta, esta tarefa rapidamente se tornará penosa, e propensa a introduzir defeitos, ocasionando retrabalho. A esta proliferação de versões, Foote (1988) denomina metastização, imprimindo uma conotação pejorativa para indicar um método repleto de problemas, que acresce dificuldades à tarefa de manutenção, desencadeando sérios problemas de gerenciamento de versão.

3.1.2 Planejamento

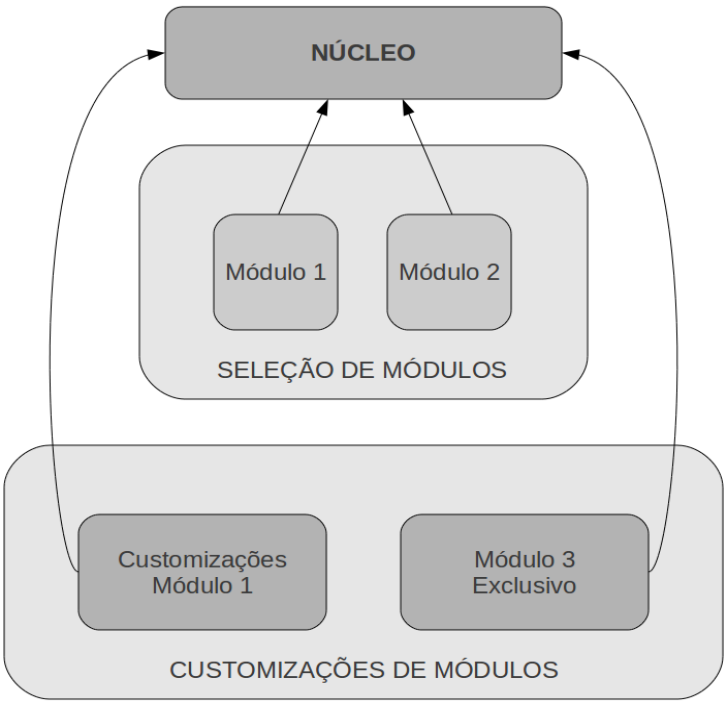
Tendo em vista as necessidades da organização em questão e as técnicas apresentadas, nesta seção são delineadas duas ações que podem ser tomadas para abordar o problema apresentado.

A primeira ação a ser tomada é definir os aspectos que estão mais propensos a alterações, e mapeá-los por modelos, de forma que possam ser manipulados posteriormente para permitir customizações. Neste trabalho, estes aspectos correspondem às interfaces gráficas, e eventuais classes que estejam relacionadas a elas, responsáveis pela persistência dos dados. No caso das interfaces, elas serão geradas em tempo de execução a partir dos modelos; para que as classes acompanhem as mudanças feitas nas interfaces, seus atributos serão descritos por um modelo, de forma que possam ser modificados por meio de reflexão computacional.

Como visto na revisão bibliográfica, é importante manter o código customizado separado do original, assim como customizações sendo feitas por meio de adições e extensões em vez de alterações no código original (BARSTOW; ARANGO, 1991; FOOTE, 1988). A proliferação de versões pode ser solucionada pelo emprego de uma arquitetura que permita alterações de funcionalidades por meio de adições e extensões, e ainda permita criar um módulo comum para todos os clientes, ao mesmo tempo possibilitando criar um módulo customizado que atenda as especificidades de cada cliente, buscando-se manter as customizações isoladas do sistema padrão, num módulo independente. Assim, o código customizado é adicionado dinamicamente por meio da reflexão computacional e geração de código, visando reincorporar automaticamente as alterações feitas no sistema original aos sistemas customizados.

As customizações identificadas na análise de lacunas são desenvolvidas como módulos, e instaladas separadamente. Isso é possível em decorrência dos principais componentes da arquitetura: o gerenciador de módulos, responsável pela instalação e inicialização de cada um dos módulos, e o núcleo, que desempenha um papel essencial para que funcionalidades sejam carregadas e customizadas, armazenando uma série de informações relativas a cada funcionalidade. A figura 4 apresenta a visão de alto nível dos principais componentes da arquitetura, suas relações e o seu papel no processo de customização.

Figura 4 – Funcionamento esperado do mecanismo no processo de customização



Fonte: Desenvolvimento próprio, 2011

Para que o macanismo funcione como esperado, é necessário definir a estrutura dos módulos e como seriam armazenados e instalados no sistema. Para permitir customizações, os modelos podem ser combinados pelo núcleo por meio de regras de precedência, permitindo que as características sejam modificadas de acordo com as necessidades.

3.2 TOMADA DE AÇÃO

Com base no diagnóstico, foram analisados os elementos mais comuns utilizados nas interfaces gráficas de sistemas web, e que foi criado um modelo permita descrevê-los. O modelo foi estruturado utilizando a linguagem YAML (YAML Ain't Markup Language), que, segundo Carneiro Júnior et al. (2010), é uma linguagem especial para representar objetos em texto simples, utilizando espaços para transmitir a estrutura e significado. Assim como o XML, pode ser usada com qualquer linguagem de programação, entretanto é mais simples. O modelo permite descrever elementos como formulários, campos para entrada de dados (com vários atributos, como máscaras, validações etc.), links, botões,

tabelas e outros elementos visuais, como abas. Nas classes responsáveis pela persistência dos dados, o modelo abrange os atributos, mapeando colunas no banco de dados, relações com outras tabelas, e validações. Na sequência, serão detalhados os modelos propostos.

3.2.1 Definição das Classes de Persistência

Em uma arquitetura MVC, os modelos são responsáveis pela persistência dos dados. Assim, é necessário que suas características sejam configuráveis para acompanhar as alterações nas interfaces com o usuário, e por isso foi criada uma maneira para descrevê-los. Cada modelo é definido pela palavra-chave model, e suas características podem ser especificadas por meio dos atributos descritos na tabela 3.

Tabela 3 – Atributos utilizados para descrever os modelos

Atributo	Descrição	Obrigatório
fields	Informações sobre os campos do modelo, em que os dados são armazenados	Sim
associations	Definição das associações com outros modelos	Não
validates_with	Classe responsável por validar o modelo	Não

Fonte: Desenvolvimento próprio, 2011

Cada modelo possui vários campos, onde suas informações são armazenadas. Eles são descritos por meio dos atributos listados na tabela 4. Também podem ser especificados relacionamentos com outros modelos. Finalmente, é possível associar uma classe cuja responsabilidade é validar os dados do modelo; esta característica será explorada no tópico que trata das validações.

Tabela 4 – Atributos utilizados para descrever os campos de um modelo

Atributo	Descrição	Obrigatório
label	Nome que será apresentado ao usuário por padrão	Sim
name	Nome da coluna	Sim
default_value	Valor padrão do campo	Não
type	Tipo do campo. Valores possíveis: string, text, boolean, integer, decimal, password, date, time, file	Sim
convert_with	Classe responsável por fazer a conversão da informação do formulário para o objeto	Não. Padrão de acordo com tipo
validations	Validações a serem aplicadas ao campo	Não
remove	Utilizado para remover o campo. Valores possíveis: true, false	Não. Padrão false

Fonte: Desenvolvimento próprio, 2011

O atributo convert_with está relacionado à conversão de dados, que é necessária pois as informações digitadas pelos usuários são recebidas como texto pelo servidor, e devem ser convertidas para o tipo correto da coluna. Por padrão, a conversão é feita de acordo com o tipo do campo, entretanto, em alguns casos é necessário especificar como a conversão será feita por meio do atributo convert_with, especialmente no caso de valores que são digitados com formatação e armazenados sem formatação. Uma classe responsável pela conversão deve implementar os métodos from_string e as_string, que convertem de texto para objeto e de objeto para texto, respectivamente.

3.2.1.1 Validações

É possível definir validações para cada campo, e também para validar o modelo todo, levando em consideração todos os seus dados. é possível informar a classe responsável pela validação na opção validates_with, de forma que a classe deve implementar o método validate(registro). O modelo proposto permite que validações sejam efetuadas individualmente em cada campo. As validações existentes para campos são descritas na tabela 5.

Tabela 5 – Atributos disponíveis para validação

Validação	Descrição	Opções
required	Campo obrigatório	
unique	Valor do campo deve ser único	
number	Valores numéricos	min, max, only_integer
exclusion	O valor não deve estar presente na lista	
inclusion	O valor deve estar presente na lista	
format	O valor deve possuir o formato descrito pela expressão regular	
length	O comprimento deve respeitar determinados critérios	min, max, equal
custom	Validação customizada	

Fonte: Desenvolvimento próprio, 2011

É importante destacar que os validadores customizados de campos devem ser classes que implementem o método validate_each (objeto, atributo, valor).

3.2.1.2 Associações

Também é possível definir associações ou relacionamentos entre modelos, do tipo 1:1, 1:N e N:M. Os tipos de relacionamento possíveis são belongs_to, has_one e has_many. Os tipos belongs_to e has_one são utilizados para mapear relações 1:1, enquanto has_many representa uma relação 1:N. Relações do tipo N:M são suportadas por meio da opção through na relação has_many. A tabela 6 sumariza as opções necessárias para descrever uma associação.

Tabela 6 – Atributos utilizados para descrever associações

Atributo	Descrição	Obrigatório
name	Nome da associação	Sim
type	Tipo da associação Valores possíveis: belongs_to, has_one, has_many	Sim
foreign_key	Chave estrangeira	Sim
class_name	Classe/Modelo cujo relacionamento mapeia	Sim
through	Permite criar uma relação a partir de outra relação, usado para construir relações N:M	Não. Presente apenas em relações do tipo has_many

Fonte: Desenvolvimento próprio, 2011

Apenas o tipo has_many possui um atributo extra, through, usada para criar uma associação do tipo N:M com outro modelo. Este tipo de associação indica que o modelo pode ser relacionado com zero ou mais instâncias do outro modelo por meio de uma associação, especificada na opção through. No quadro 1 é mostrado um exemplo de definição de associações.

Quadro 1 – Exemplo de descrição de associações

```
associations:
- name: cliente
  label: “Cliente”
  type: belongs_to
    foreign_key: cliente_id
    class_name: Pessoa
- name: pedido_produtos
  type: has_many
    foreign_key: pedido_id
    class_name: PedidoProduto
- name: produtos
  type: has_many
    foreign_key: produto_id
    class_name: Produto
    through: pedido_produtos
```

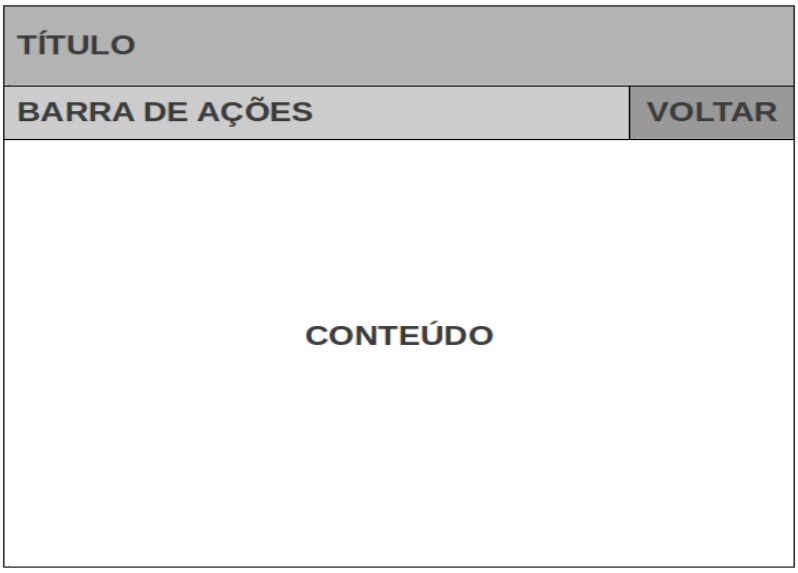
Fonte: Desenvolvimento próprio, 2011

As definições dos modelos são armazenadas no diretório app/models de cada módulo, possuem a extensão .def, são lidas ao inicializar o sistema, e aplicadas aos modelos por meio de reflexão.

3.2.2 Definição das Interfaces

Para lidar com alterações nas interfaces com o usuário, também foi desenvolvido um modelo para descrevê-las utilizando a linguagem YAML. Como os principais elementos encontrados em sistemas web são formulários, tabelas e relatórios, o modelo visa atender a essas especificidades, entretanto é possível que os modelos apresentados sejam incrementados por meio da adição de novos atributos, para suportar outros componentes de acordo com a demanda. O modelo traz ainda como vantagem a padronização das telas de todo o sistema, fazendo com que os elementos mantenham sempre a mesma aparência, espaçamento, cores, tamanho etc, simplificando o desenvolvimento e facilitando futuras alterações dos aspectos visuais da interface. Para a elaboração desta proposta, considera-se que as interfaces possuem os elementos vistos na figura 5, na disposição apresentada. Entretanto, é possível adicionar novos elementos ou adaptar o modelo de acordo com a necessidade.

Figura 5 – Disposição dos elementos principais no sistema



Fonte: Desenvolvimento próprio 2011

No topo é exibido o título da página, que é obrigatório e indica qual é a funcionalidade acessada no momento. Abaixo dele estão a barra de ações e um link de voltar, sendo que no primeiro estão presentes as ações possíveis de execução a partir desta página, e o link de voltar para a página anterior. O restante da tela é dedicado ao conteúdo da página, onde são exibidas informações ao usuário, permitindo interações. Os atributos para adicionar conteúdo neste formato são apresentados na tabela 7.

Tabela 7 – Atributos para adicionar conteúdos à interface

Atributo	Descrição	Obrigatório
title	Título da página	Sim
actions	Barra de ações: contém ações que podem ser realizadas a partir da página	Não
back	URL de retorno	Não
inherit_from	Utilizado em sobreposições, permite que sejam herdados atributos de outra definição	Não

Fonte: Desenvolvimento próprio, 2011

A barra de ações contém apenas links para outras funcionalidades, que podem ser definidos com os atributos descritos na tabela 8.

Tabela 8 – Atributos utilizados para descrever links

Atributo	Descrição	Obrigatório
label	Texto que será apresentado ao usuário	Sim
url	URL apontada pelo link	Sim
image	Imagem para ser exibida junto ao link	Não
hint	Texto explicativo exibido ao passar o mouse sobre o link	Não

Fonte: Desenvolvimento próprio, 2011

É importante notar que para todos os elementos apresentados nesta proposta é possível especificar outros atributos HTML pertinentes ao que está sendo descrito, tais como id, class etc. Outros elementos especificados na definição são adicionados diretamente na área reservada ao conteúdo da página.

3.2.2.1 Definição de formulários

Como visto, um dos principais componentes de sistemas web são formulários e tabelas. Um formulário pode ser definido a partir da chave form, e podem ser utilizados os atributos da tabela 9.

Tabela 9 – Atributos utilizados para descrever formulários

Atributo	Descrição	Obrigatório
action	URL para onde os dados serão submetidos	Sim
fields	Campos que serão submetidos	Sim

Fonte: Desenvolvimento próprio, 2011

Um formulário pode possuir vários campos para entrada de dados, e para cada um, é possível especificar os atributos apresentados na tabela 10.

Tabela 10 – Atributos utilizados para descrever campos

Atributo	Descrição	Obrigatório
label	Nome que será apresentado ao usuário	Não
name	Nome da coluna	Sim
hint	Ajuda para preenchimento do campo	Não. Padrão vazio
mask	Apoio para entrada de dados por meio de formatação, como datas, horas, CPF etc	Não
hidden	Indica se o campo será visível ao usuário. Valores possíveis: true, false	Não. Padrão false
rendered	Indica se o campo será renderizado. Valores possíveis: true, false	Não. Padrão true
disabled	Indica se o campo será desabilitado, impedindo o usuário de alterá-lo. Valores possíveis: true, false	Não. Padrão false
convert_with	Classe responsável por fazer a conversão da informação do formulário para o objeto	Não
break_before	Força a inserção de uma quebra de linha antes do elemento.	Não. Padrão false
insert_after	Utilizado em sobreposições para posicionar o elemento após um outro	Não

insert_before	Utilizado em sobreposições para posicionar o elemento antes de outro	Não
inser_in_group	Utilizado em sobreposições para posicionar o elemento dentro de um grupo	Não
subfields	Utilizada para especificar os campos de relações do tipo has_many	Não
static_collection	Para campos em que exista opção de escolha de valores, podem ser especificados por meio deste atributo, no formato [chave, valor]	Não
filtered_collection	Para campos em que exista opção de escolha de valores, mas exista uma grande quantidade de valores e exista a necessidade de filtragem	Não

Fonte: Desenvolvimento próprio, 2011

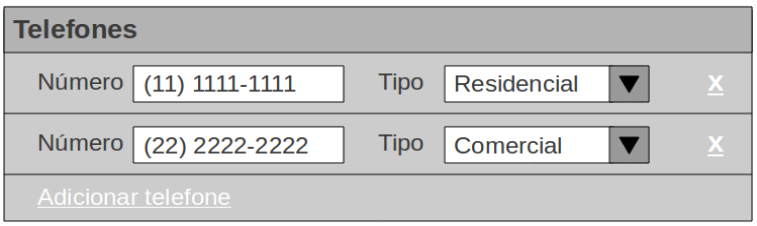
Na tabela 10, é importante ressaltar alguns pontos. A definição subfields é utilizada para especificar quais campos serão exibidos para uma relação do tipo has_many. Essa declaração permite que registros sejam adicionados e removidos dinamicamente por meio de JavaScript, valendo-se de um template oculto. No quadro 2 é apresentado um exemplo de uma declaração subfields e sua representação gráfica pode ser vista na figura 6.

Quadro 2 – Exemplo de declaração subfields

```
fields:
- name: Telefones
  subfields:
  - name: numero
    mask: “(##) ####-####”
  - name: tipo
    static_collection: [ [1, “Residencial”], [2,
“Comercial”] ]
```

Fonte: Desenvolvimento próprio, 2011

Figura 6 – Exemplo de interface gerada pela declaração subfields



Fonte: Desenvolvimento próprio, 2011

É importante notar que o atributo name suporta uma sintaxe especial para acessar valores em relações do tipo belongs_to, no formato relação#método. É feita a inferência de qual campo deve ser populado por meio da chave estrangeira definida na relação, e a apresentação do valor é feita por meio do método estipulado. Este tipo de relação ainda permite que seja especificada uma coleção para a seleção dos valores de duas maneiras. A primeira é a coleção estática, em que uma lista fixa de elementos é informada para seleção. Uma coleção estática recebe os valores da coleção diretamente, ou interpreta algum código para gerar a coleção no formato esperado. A segunda maneira possível é uma coleção filtrada, em que os dados serão obtidos a partir de uma URL no formato correto. Conforme o usuário digita no campo, os valores vão sendo filtrados para tornar mais fácil a seleção em um grande número de registros.

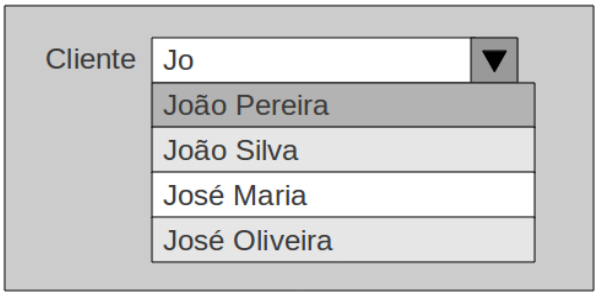
O quadro 3 exibe um exemplo da utilização, e a aparência no sistema é mostrada na figura 7.

Quadro 3 – Exemplo de utilização de coleção filtrada

```
- name: cliente
  filtered_collection:
    data_url: “/pessoas/index”
    param_name: nome
     colums:
     - name: nome
```

Fonte: Desenvolvimento próprio, 2011

Figura 7 – Exemplo de interface gerada pela coleção filtrada



Fonte: Desenvolvimento próprio, 2011

É feita uma requisição à URL informada em data_url, enviando como parâmetro param_name o valor digitado no campo, e são retornados os valores filtrados no formato JSON (JavaScript Object Notation). Após o retorno dos dados, são exibidas na lista de valores possíveis as colunas informadas em columns.

Botões são outro elemento muito comum em aplicações web, e são definidos por meio da chave button. Os atributos possíveis são descritos na tabela 11.

Tabela 11 – Atributos utilizados para descrever botões

Atributo	Descrição	Obrigatório
name	Identificador do botão	Sim
label	Texto que será apresentado dentro do botão	Sim

Fonte: Desenvolvimento próprio, 2011

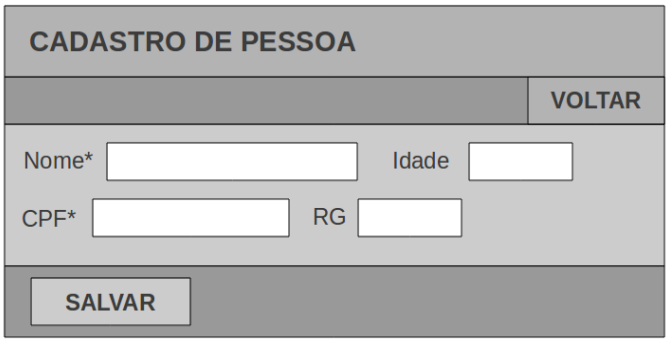
Com relação à aparência do sistema, é importante destacar que cada um dos tipos apresentados na tabela 4 irá gerar um campo específico, mantendo um padrão em todo o sistema. Também vale notar que os elementos são posicionados de maneira fluída, de forma a se adaptar à resolução do usuário. Serão alinhados tantos campos quantos forem possíveis, a menos que seja forçada a criação de uma nova linha para um campo por meio do atributo break_before. No quadro 4 é apresentado um exemplo de formulário, e na figura 8 como o mesmo é apresentado.

Quadro 4 – Exemplo da definição de um formulário com o modelo proposto

```
title: Cadastro de pessoa
back: “/pessoas/index”
form:
  action: “/pessoas/create”
  id: form_pessoa
  fields:
  - name: nome
  - name: idade
  - name: cpf
    mask: ###.###.###-##
  - name: rg
  button:
    name: salvar
    label: Salvar
```

Fonte: Desenvolvimento próprio, 2011

Figura 8 – Exemplo de formulário gerado pelo modelo proposto



Fonte: Desenvolvimento próprio, 2011

É importante atentar que alguns dados são obtidos por meio de reflexão, por exemplo, quando um atributo possui validação de obrigatório, é exibido um asterisco após sua descrição para indicar ao usuário.

3.2.2.2 Grupos e abas

Muitas vezes, quando há uma grande quantidade de campos, existe a necessidade de prover uma melhor disposição deles, visando simplificar o preenchimento do formulário. Desta forma, o modelo proposto permite que campos sejam agrupados para melhorar a organização do formulário e facilitar seu preenchimento. Os campos podem ser

divididos em abas, ou mesmo agrupados. Um grupo reúne campos que possuem alguma relação semântica, por meio da chave group, utilizando um cabeçalho para identificar o tipo de informação, e possui os atributos listados na tabela 12. Na figura 9 é mostrado um exemplo de dois grupos, gerado pela definição vista no quadro 5.

Tabela 12 – Atributos utilizados para agrupar elementos em um formulário

Atributo	Descrição	Obrigatório
name	Identificação do grupo	Sim
label	Nome que será exibido no cabeçalho do grupo	Sim
visible	Indica se o grupo está visível Valores possíveis: true, false	Não. Padrão true
fields	Campos contidos dentro do grupo. Podem ser especificados por meio dos atributos descritos na tabela 10.	Sim
ungroup	Utilizado em sobreposições para fazer com que os elementos do grupo sejam desagrupados Valores possíveis: true, false	Não

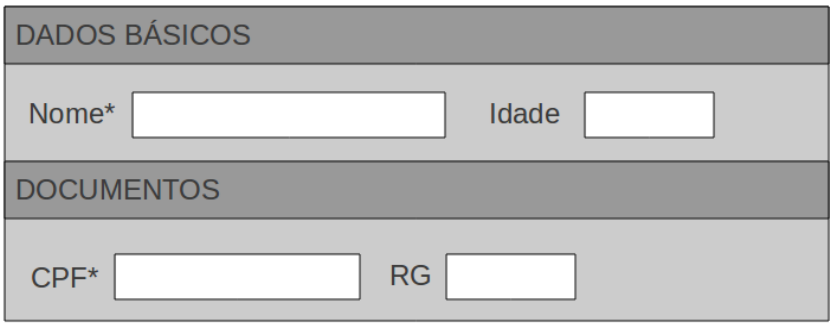
Fonte: Desenvolvimento próprio, 2011

Quadro 5 – Exemplo de definição de agrupamento de campos em um formulário

```
group:
  name: dados_basicos
  label: Dados básicos
  fields:
  - name: nome
  - name: idade
group:
  name: documentos
  label: Documentos
  fields:
  - name: cpf
  - name: rg
```

Fonte: Desenvolvimento próprio, 2011

Figura 9 – Exemplo de interface gerada usando agrupamento de campos



Fonte: Desenvolvimento próprio, 2011

Em alguns casos, quando a quantidade de informação a ser preenchida é grande, pode ser interessante manter uma separação visual para que o usuário mantenha o foco em um grupo de campos. Neste sentido, é provido o conceito de aba, que atua de forma semelhante ao grupo, com a diferença que apenas o conteúdo de uma aba é exibido por vez. Uma aba pode conter vários grupos, e ao ser acionada são exibidos apenas os seus campos. É identificada pela chave tab, e pode conter os atributos apresentados na tabela 13. No quadro 6 é exibido um exemplo de divisão em abas, e na figura 10 o resultado.

Tabela 13 – Atributos utilizados para separar campos de um formulário em abas

Atributo	Descrição	Obrigatório
name	Identificador da aba	Sim
label	Nome que será exibido na aba	Sim
visible	true|false	Não. Padrão true
fields	Campos contidos dentro da aba. Podem ser especificados por meio dos atributos descritos na tabela 10.	Sim

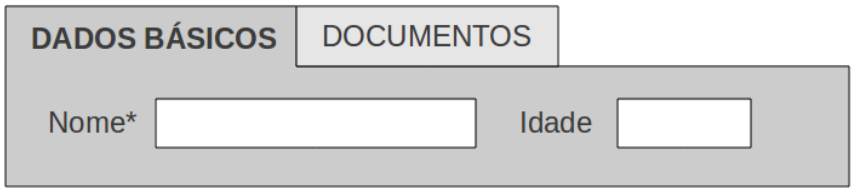
Fonte: Desenvolvimento próprio, 2011

Quadro 6 – Exemplo de definição de agrupamento de campos em abas

```
tab:
  name: dados_basicos
  label: Dados básicos
  fields:
  - name: nome
  - name: idade
tab:
  name: documentos
  label: Documentos
  fields:
  - name: cpf
  - name: rg
```

Fonte: Desenvolvimento próprio, 2011

Figura 10 – Exemplo de aba gerada



Fonte: Desenvolvimento próprio, 2011

3.2.2.3 Tabelas

Tabelas são um elemento bastante utilizado para exibição de dados. No modelo proposto podem ser de dois tipos, estática e filtrada. A tabela estática serve para exibir uma coleção fixa de objetos, identificada pela chave static_table, já a filtrada permite que os dados sejam filtrados, ordenados e paginados. Tabelas estáticas podem possuir os atributos descritos na tabela 14.

Tabela 14 – Atributos utilizados para descrever tabelas estáticas

Atributo	Descrição	Obrigatório
name	Identificador da tabela	Sim
data_source	Fonte de dados que vai popular a tabela	Sim
columns	Colunas da tabela	Sim
footer	Rodapé, para informações adicionais	Não

Fonte: Desenvolvimento próprio, 2011

As colunas de uma tabela estática podem possuir os atributos listados na tabela 15, e são exibidas na ordem em que são definidas.

Tabela 15 – Atributos utilizados para descrever colunas de uma tabela estática

Atributo	Descrição	Obrigatório
name	Representa o atributo que contém o valor desejado na coleção	Sim
label	Texto que será apresentado no cabeçalho para identificar a coluna	Sim
visible	Valores possíveis: true, false	Não. Padrão true
convert_with	Classe responsável por converter o valor	Não. Padrão é obtido a partir do tipo do atributo

Fonte: Desenvolvimento próprio, 2011

As tabelas filtradas são identificadas pela chave filtered_table podem possuir os atributos apresentados na tabela 16.

Tabela 16 – Atributos utilizados para descrever tabelas filtradas

Atributo	Descrição	Obrigatório
name	Identificador da tabela	Sim
url	URL de onde os dados serão filtrados	Sim
columns	Colunas da tabela	Sim
footer	Rodapé, para informações adicionais	Não
search_form	Formulário para filtragem	Não
per_page	Quantidade de dados por página	Não. Padrão 30
default_sort	Ordenação padrão Exemplo: name ASC	Sim

Fonte: Desenvolvimento próprio, 2011

As colunas neste tipo de tabela podem possuir os atributos apresentados na tabela 17.

Tabela 17 – Atributos utilizados para descrever colunas de uma tabela filtrada

Atributo	Descrição	Obrigatório
name	Representa o atributo que contém o valor desejado na coleção	Sim
label	Texto que será apresentado no cabeçalho para identificar a coluna	Sim
visible	Valores possíveis: true, false	Não. Padrão true
convert_with	Classe responsável por converter o valor para exibição	Não. Padrão é obtido a partir do tipo de dado
sortable	Indica se é possível ordenar a tabela pela coluna Valores possíveis: true, false	Padrão true

Fonte: Desenvolvimento próprio, 2011

Uma tabela filtrada é acompanhada de um formulário para a filtragem dos dados. Para tanto é utilizada a chave search_form, que pode possuir os atributos apresentados na tabela 18. No quadro 7 pode ser visto um exemplo de descrição de tabela filtrada. A figura 11 exibe a tabela gerada, como vista pelo usuário.

Tabela 18 – Atributos utilizados para descrever um formulário de busca

Atributo	Descrição	Obrigatório
title	Título da filtragem	Não. Padrão “Buscar por:”
name	Nome da coluna para filtragem	Sim
label	Texto exibido ao usuário	Sim
type	Tipo da coluna. Valores possíveis: string, date, integer, decimal	Sim
filter_button_label	Texto exibido no botão de filtragem	Não. Padrão “Buscar”
clear_button_label	Texto exibido no botão de limpar filtragem	Não. Padrão “Limpar”

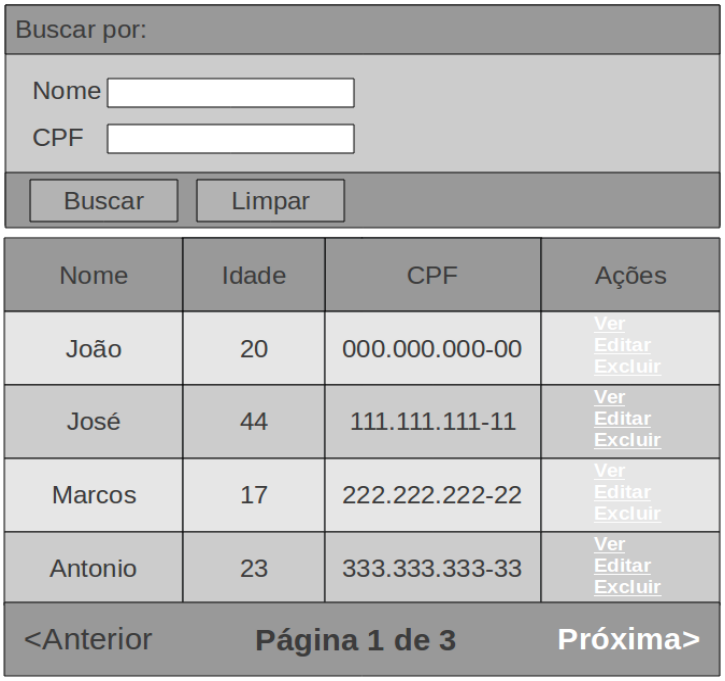
Fonte: Desenvolvimento próprio, 2011

Quadro 7 – Exemplo de definição de tabela filtrada

```
filtered_table:
  url: “/pessoas/index”
  per_page: 4
  columns:
  - name: nome
  - name: idade
  - name: cpf
    sortable: false
  - name: acoes
    label: Ações
    sortable: false
    convert_with: LinkGenerator
  search_form:
    fields:
    - name: nome
    - name: cpf
    filter_button_label: Buscar
    clear_button_label: Limpar
```

Fonte: Desenvolvimento próprio, 2011

Figura 11 – Exemplo de tabela filtrada



Fonte: Desenvolvimento próprio, 2011

3.2.2.4 Visualização dos dados

A exibição de dados no sistema é feita utilizando a chave show, que possui os atributos descritos na tabela 19. No quadro 8 pode ser visto um exemplo de utilização e a figura 12 ilustra como as informações são exibidas ao usuário.

Tabela 19 – Atributos utilizados para descrever informações para visualização

Atributo	Descrição	Obrigatório
source	Fonte de dados a ser exibida	Sim
fields	Campos que serão exibidos. Podem ser especificados por meio dos atributos descritos na tabela 10.	Sim

Fonte: Desenvolvimento próprio, 2011

Quadro 8 – Exemplo de descrição da visualização dos dados

```
title: Visualizando Pessoa
show:
  source:- @pessoa
  fields:
  - name: nome
  - name: idade
  - name: cpf
  - name: telefones
    subfields:
    - name: numero
    - name: tipo
      convert_with: TipoConverter
```

Fonte: Desenvolvimento próprio, 2011

Figura 12 – Exemplo de visualização dos dados

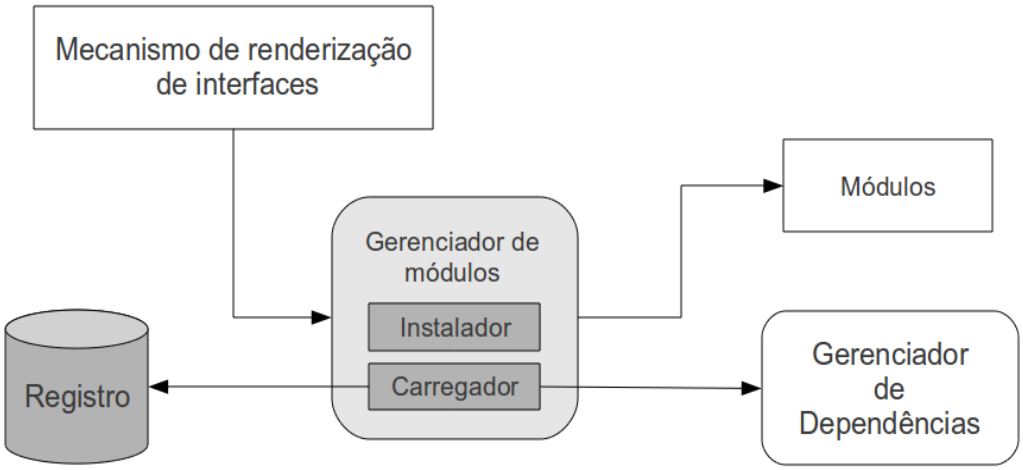
Fonte: Desenvolvimento próprio, 2011

De forma similar, a geração de relatórios é feita por meio da chave report, que possui as mesmas propriedades que a chave show. A diferença encontra-se no layout aplicado, que no caso de relatórios é ideal para impressão.

3.2.3 Estruturação em Módulos

Após a criação dos modelos, a segunda ação tomada foi estruturar a arquitetura de forma que cada módulo fosse transformado em uma aplicação indpendente, para manter customizações isoladas das funcionalidades originais. A arquitetura baseia-se no trabalho de Cruz (2007), que permite que funcionalidades sejam adicionadas e removidas no formato de módulos. A visão geral da arquitetura é apresentada na figura 13.

Figura 13 – Visão geral dos componentes da arquitetura proposta



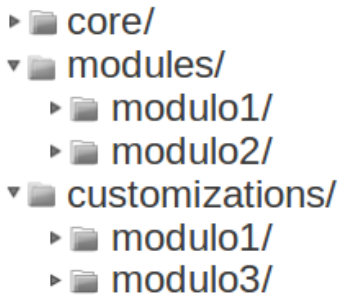
Fonte: Desenvolvimento próprio, 2011

Em primeiro lugar, o registro é essencialmente um objeto global com uma interface que descreve suas operações, possibilitando que informações sobre os módulos sejam armazenadas e obtidas em tempo de execução. Este componente é implementado utilizando o padrão REGISTRY descrito por Fowler et. al. (2002) como um ponto de acesso central que outros objetos podem usar para encontrar serviços comuns, e armazena os dados em uma tabela hash, de forma a agilizar a recuperação de valores.

O gerenciador de dependências foi implementado usando o padrão PLUGIN (FOWLER, 2010). A ordem de registro não importa, pois nenhum serviço é criado até que tudo esteja registrado. O gerenciador de dependências se preocupa para garantir que tudo seja criado até o momento de efetiva utilização. Depois de registrados os objetos necessários, basta pedir ao gerenciador de dependências que instancie a interface desejada, e ele se encarregará de montar o objeto a partir dos componentes registrados. Assim, o gerenciador de dependências é o responsável pela injeção de dependências, desacoplando o código e permitindo maior flexibilidade, permitindo selecionar implementações de um contrato.

O gerenciador de módulos ocupa um papel central no mecanismo proposto e possui dois componentes principais, o instalador e o carregador de módulos. O primeiro ponto relacionado à proposta desenvolvida diz respeito à localização em que os módulos serão instalados. A estrutura de diretórios proposta pode ser observada na figura 14.

Figura 14 – Estrutura geral de diretórios do sistema proposto



Fonte: Desenvolvimento próprio, 2011

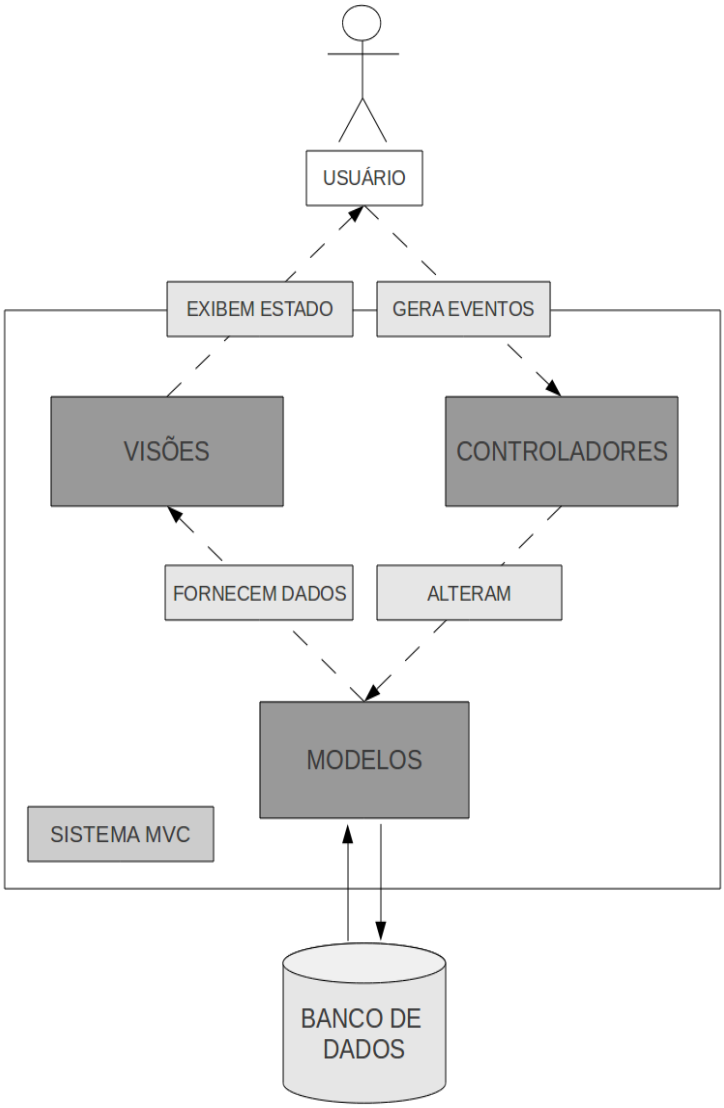
O núcleo do sistema, situado no diretório core, fornece recursos que serão utilizados por todo o sistema. No diretório modules, são armazendos os módulos originais do sistema e em customizations situam-se as customizações. Com esta organização, torna-se possível isolar o código customizado do original, de forma que o gerenciador de módulos fique responsável por combinar modelos e decidir quais configurações serão utilizadas em tempo de execução.

3.3 Estrutura dos Módulos

Um módulo, no contexto desta proposta é considerado como uma aplicação independente, que segue uma estrutura de arquivos pré-definida, e empacotado em um arquivo único. Cada módulo é uma aplicação que segue a arquitetura MVC (Model-View-Controller), que segundo Cicchetti et al. (2007), é um padrão de arquitetura que visa minimizar o grau de acoplamento entre a interface de usuário e os modelos do domínio de forma eficaz. Cada vez mais, esse padrão é utilizado no desenvolvimento de programas com linguagens orientadas a objeto e para organizar projetos de aplicações Web, pois defende uma divisão da lógica da aplicação em três categorias distintas: o modelo, visão e controlador. O modelo representa os dados, a visão representa a interface do usuário, e o controlador gerencia todas as ações. Cada uma destas responsabilidades é independente, uma mudança em um modelo não precisa afetar as visões; igualmente, uma mudança em uma visão não deve ter efeito sobre o modelo. Isto significa que as mudanças em uma aplicação MVC tendem a ser localizadas e de baixo impacto, aliviando consideravelmente o esforço de manutenção, aumentando o nível de reutilização dos componentes (CARNEIRO JÚNIOR et al, 2010).

Conforme Carneiro Júnior et. al, (2010), o controle de fluxo no ciclo MVC geralmente funciona como segue: a) O usuário interage com a interface e dispara um evento (por exemplo, envia um formulário); b) O controlador recebe a entrada da interface (por exemplo, os dados enviados pelo formulário); c) O controlador acessa o modelo, muitas vezes, atualizando-o de alguma maneira (por exemplo, criar um novo registro com os dados do formulário); d) O controlador invoca uma visão que transmite uma interface atualizada; e) A interface espera por outras interações do usuário, e o ciclo se repete. A figura 15 ilustra estas interações.

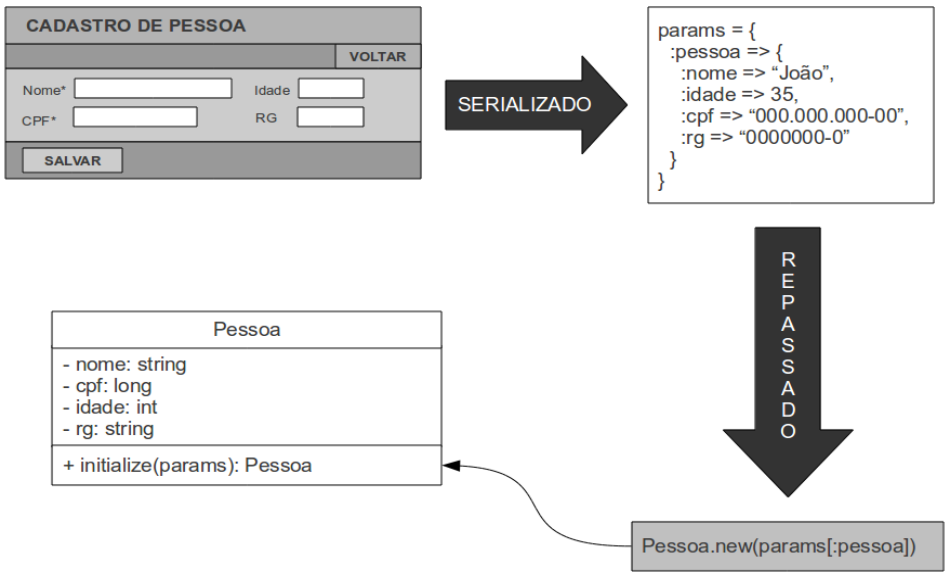
Figura 15 – Interações entre camadas no ciclo MVC



Fonte: Adaptado de Carneiro Júnior et al .(2010, p. 9)

Na figura 16 é exibido o fluxo das informações, quando um formulário é enviado, as informações contidas nos campos da interface gráfica são serializados em uma tabela hash, recebidas pelo controlador, que repassa ao contrutor da classe responsável pela persistência dos dados. Esta instancia um novo objeto com os dados recebidos por meio de reflexão.

Figura 16 – Fluxo de informações entre os componentes



Fonte: Desenvolvimento próprio, 2011

Para que o fluxo descrito seja executado adequadamente, faz-se necessária uma camada de mapeamento objeto-relacional (ORM – Object-Relational Mapping), de forma que a classe seja mapeada para uma tabela do banco de dados. O construtor desta classe deve receber como parâmetro um único argumento, uma tabela hash, em que as chaves correspondem aos nomes das colunas, para permitir que todos os campos sejam populados ao mesmo tempo. É importante notar que os campos na interface devem ser nomeados de acordo com os nomes dos atributos descritos no modelo da classe, para permitir que cada um dos valores sejam atribuídos ao respectivo atributo por meio de reflexão. Deste modo, tendo em vista o padrão MVC, os módulos são estruturados como visto na figura 17.

Figura 17 – Estrutura de diretórios de um módulo

Fonte: Desenvolvimento próprio, 2011

Dentro do diretório app é armazenado o código da aplicação. É subdividido em controllers, models e views, que são responsáveis pelo fluxo da aplicação, regras de negócio e telas do sistema, respectivamente. As telas são especificadas por meio das definições de interface, que foram vistas anteriormente. O diretório models armazena, além das regras de negócio, as definições das classes responsáveis pela persistência dos dados, que são carregadas por meio de reflexão. O diretório db contem arquivos que efetuam as alterações necessárias no banco de dados para o funcionamento correto do módulo. O diretório public por sua vez, armazena a camada visual da aplicação, imagens, folhas de estilos, e JavaScripts.

Como um módulo não pode acessar diretamente funcionalidades de outro, ele precisa expor uma interface pública para permitir que outros módulos se comuniquem, visando estabelecer um contrato, provendo encapsulamento e minimizando interdependências entre eles. É sugerido que esta interface seja definida como um FACADE no arquivo interface, elencado por Gamma et al., (1994), expondo seus serviços e estabelecendo um conjunto de operações com que os outros módulos podem interagir. Para melhorar o isolamento entre os módulos, foram utilizados os princípios discutidos na revisão acerca de modularidade, visando ocultar detalhes de implementação. Também são necessários alguns metadados sobre o módulo, para descrever sua versão, dependências e outras informações necessárias para instalação. Finalmente, o manifesto é um script que armazena as configurações que são carregadas durante a inicialização do sistema; interage com o registro do sistema para informá-lo das funcionalidades do módulo e como elas estarão disponíveis aos usuários.

Existe ainda um arquivo opcional nesta estrutura apresentada, que deve estar presente apenas em módulos customizados. Este arquivo deve possuir o nome customization.txt, e tem como propósito único diferenciar uma customização dos módulos originais do sistema durante a instalação. Apesar do conteúdo deste arquivo não ser importante, a ele podem ser adicionadas informações sobre quais alterações são feitas pelo módulo, servindo como documentação para os desenvolvedores. Após estruturar o módulo, é necessário empacotá-lo utilizando algum formato, tornando-o um pacote único, para facilitar distribuição.

3.3.1 Instalação e Registro dos Módulos

Quando um módulo é selecionado para instalação, o instalador de módulos o descompacta dentro do diretório modules (ou customizations caso possua o arquivo customization.txt), e executa o processo de instalação, que é responsável por fazer as alterações necessárias na estrutura do banco de dados, valendo-se das informações contidas no diretório db de cada módulo. Neste processo, são efetuadas as adequações necessárias no banco de dados; tabelas são criadas ou alteradas, e são armazenadas informações que sejam necessárias para o correto funcionamento do módulo, como configurações padrão e dados iniciais. É importante salientar que o processo de instalação de módulos customizados é o mesmo, com a diferença que são armazenados no diretório customizations, mesmo nos casos de módulos desenvolvidos exclusivamente para manter uma separação conceitual das funcionalidades originais do sistema.

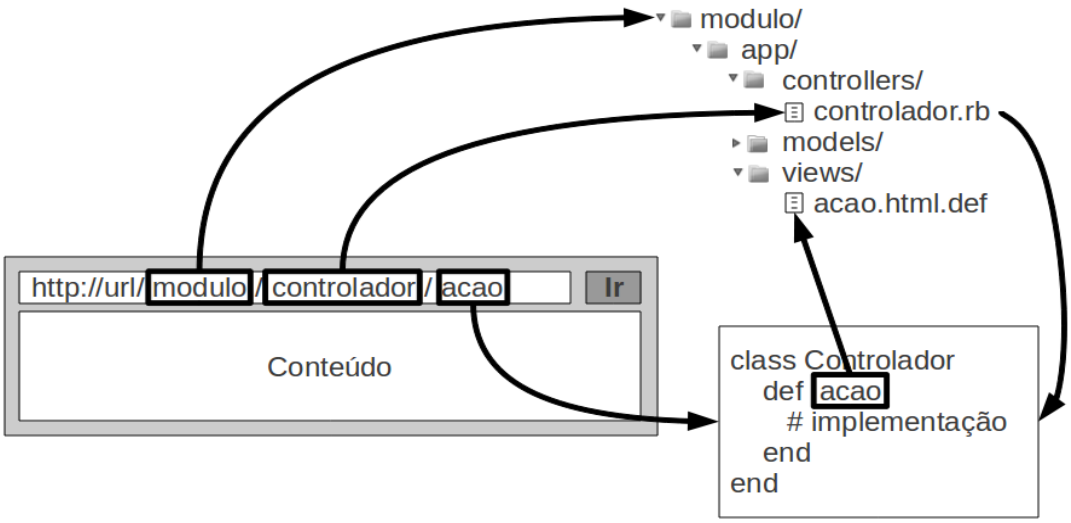
Após a instalação dos módulos, o sistema já pode ser inicializado. Nesta fase, o carregador de módulos percorre o diretório modules, e para cada um dos módulos instalados, carrega a lógica contida no diretório app, publica sua interface carregando o arquivo interface e executa o seu manifesto. O manifesto é o responsável por fazer com que o sistema tome conhecimento da existência do módulo, por meio de APIs (Application Programming Interfaces) fornecidas pelo núcleo.

É importante frisar que o procedimento de inicialização de módulos customizados é o mesmo; a diferença é que as configurações do manifesto possuem maior prioridade e sobrescrevem as definidas pelos módulos originais, modificando o comportamento do módulo original, pois são carregadas depois deles. Utilizando regras de precedência, é possível definir qual código será executado, de forma que o código customizado sobreponha o código original, sem interferir nele.

3.3.2 O Processo de Geração de Interfaces Gráficas

Com base nas definições anteriores, podemos explorar o processo de geração de interfaces. Quando uma requisição é recebida, a primeira etapa é determinar qual é o código que deve ser executado, o que é alcançado por meio da análise da URL. A partir dela, são extraídas algumas informações acerca da localização das definições de interface; a figura 18 ilustra esse mapeamento.

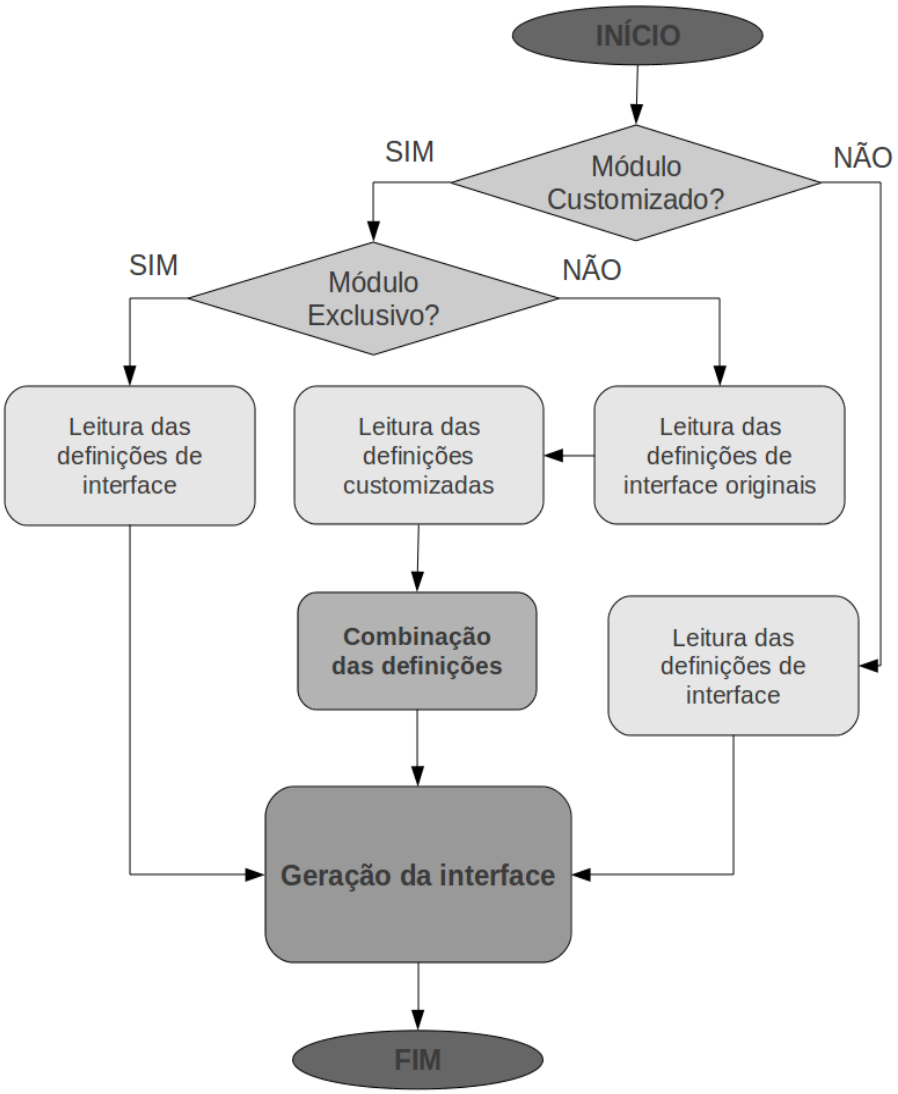
Figura 18 – Mapeamento da requisição para localização das definições de interface



Fonte: Desenvolvimento próprio, 2011

As definições de interfaces são armazenadas dentro do diretório views do módulo, e possuem a extensão .html.def, indicando um arquivo de definição que gera HTML. Como visto na figura 18, podem ser obtidos o nome do módulo, do controlador, e da ação que deve ser invocada a partir da URL. De posse destas informações, são efetuadas algumas verificações para identificar a partir de quais modelos a interface deve ser gerada. A geração é feita por meio do padrão TRANSFORM VIEW descrito por Fowler et al., (2002); a figura 19 apresenta graficamente as etapas necessárias para a geração da interface.

Figura 19 – Verificações efetuadas para determinar quais definições devem ser carregadas

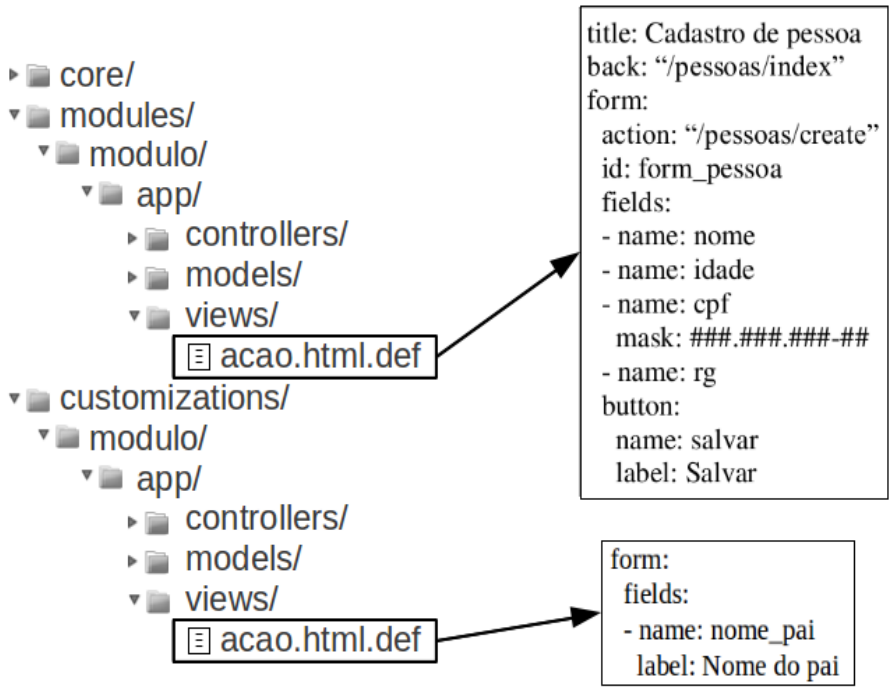


Fonte: Desenvolvimento próprio, 2011

Após a identificação da localização da ação requisitada, a primeira verificação efetuada é se a funcionalidade possui customizações, não importando se representa um módulo exclusivo ou customizações efetuadas em um módulo. Para determinar esta característica, basta investigar se o diretório customizations contém o modulo e controlador. Caso não exista customização, as definições do módulo original são lidas (localizadas em modules/modulo/controlador/acao.html.def), interpretadas, e a interface gerada a partir dela. Se existir o módulo dentro de customizations, é efetuada outra verificação para identificar se o módulo é exclusivo ou não.

Caso seja exclusivo, é suficiente ler as definições de interface (localizadas em customizations/modulo/controlador/acao.html.def) e gerar a interface. Caso contrário, será necessário ler as definições de interface originais, ler as definições customizadas, combiná-las, para só então gerar a interface. A figura 20 ilustra este caso, em que dois modelos foram encontrados para a mesma interface, e a figura 21 ilustra o modelo resultante, a partir do qual a interface será gerada.

Figura 20 – Exemplo de modelo com customização



Fonte: Desenvolvimento próprio, 2011

Figura 21 – Exemplo dos modelos combinados

```
title: Cadastro de pessoa
back: “/pessoas/index”
form:
  action: “/pessoas/create”
  id: form_pessoa
  fields:
  - name: nome
  - name: idade
  - name: cpf
    mask: ###.###.###-##
  - name: rg
  - name: nome_pai
    label: Nome do pai
button:
  name: salvar
  label: Salvar
```

Fonte: Desenvolvimento próprio, 2011

Os modelos são interpretados e armazenados numa estrutura similar a uma tabela hash, e os atributos customizados são mesclados recursivamente sobrescrevendo atributos originais. Durante a combinação de definições, a regra de precedência é simples, atributos customizados sempre tem precedência sobre os originais. A única exceção a esta regra diz respeito a campos obrigatórios já existentes em um formulário; estes não podem ser removidos, nem sua validação desativada. Entretanto, é possível adicionar novos campos obrigatórios.

3.4 CONSIDERAÇÕES

Foi apresentado um mecanismo de apoio à customização de sistemas web, com foco principal nas interfaces. Com os modelos criados, foi possível combiná-los para adicionar e remover elementos e atributos. Ficou evidenciado a necessidade de utilização de uma linguagem orientada a objeto com bom suporte à reflexão. Da mesma forma, foi constatado que é importante definir localizações onde os módulos serão instalados, bem como a detrminação da estrutura de diretórios de cada módulo e um script de inicialização, para que o módulo seja carregado junto ao sistema. Para que essa integração seja realizada, é essencial observar os requisitos para integrar o módulo ao núcleo.

3.5 TRABALHOS RELACIONADOS

Oliveira (1998) descreve uma estrutura em que as informações relativas à especificação dos sistemas são armazenadas em um repositório de dados, o metamodelo. Este mantem as especificações em alto nível de abstração, sem detalhes de implementação e em formato independente de linguagem; armazena especificações das telas que serão apresentadas ao usuário, das regras de negócio que controlam a aplicação e do banco de dados. Tais especificações podem ser convertidas para código fonte de qualquer linguagem que esteja prevista na estrutura. Se existirem módulos de conversão para novas tecnologias, a especificação inicial da aplicação não é modificada.

O metamodelo é implementado em um banco de dados relacional. O diagrama Entidade-Relacionamento proposto possui informações relativas ao diagrama de fluxo de dados (DFD), diagrama de transição de estados (DTE), regras de negócio, estrutura do banco de dados e telas da aplicação. O objetivo principal do metamodelo é permitir a especificação de sistemas de informação de maneira independente de linguagem de programação, com base em definições abstratas. Com a utilização da técnica, as regras de negócio ainda são codificadas, mas é possível atingir níveis de 98% de geração de código de interface e de banco de dados, reduzindo em mais de 50% o tempo de desenvolvimento, aumentando a produtividade e diminuindo os custos dos projetos.

Na mesma linha de pensamento, Valkov (2008) rascunha uma proposta de ERP que introduz uma plataforma baseada na colaboração global e modelos de negócio utilizando diagramas UML (Unified Modeling Language) e metadados. Esses metadados são transformados em subsistemas consistindo de objetos de dados, objetos de regra de negócio, código executável de banco de dados e interfaces de usuário. O processo de geração funciona de forma que mudanças nos modelos de negócio automaticamente são propagadas aos objetos de dados e interfaces.

Os metadados são interpretados e a partir deles são criadas estruturas de dados (tabelas, views, objetos no banco de dados e uma fina lógica de front-end) compilando objetos de negócio e gerando interfaces com o usuário em tempo de execução. A meta final é uma programação baseada em modelos por especialistas na regra de negócio, com pouca ou nenhuma necessidade de programação. A geração de interfaces é feita em tempo de execução ou durante a fase de interpretação. Todos os elementos relacionados à interface são mantidos como itens separados no banco de dados – os dados, a estrutura de apresentação, os processos

de negócios. Entretanto, até a presente data, não existe uma implementação desta plataforma, provavelmente pela sua grande complexidade (VALKOV, 2008).

Gunathunga et al. (2007), por sua vez, sugerem uma abordagem em que a interface gráfica é definida em um modelo XML (Extensible Markup Language), e gerada dinamicamente a partir dele. O modelo inclui informações sobre a aparência das interfaces de usuário, validação de dados e manipulação de eventos, e apresenta como uma das principais vantagens a dissociação da interface de usuário e lógica de negócios. A justificativa para isso é que alguns sistemas podem precisar de melhorias das suas interfaces de usuário ou implementação de interfaces totalmente novas, sem introduzir alterações na camada de negócios. Em alguns casos, um software pode manter suas interfaces de usuário existentes, ao introduzir modificações para outras camadas. Se modificações são necessárias na apresentação dos dados, todas as modificações podem ser aplicadas com a especificação XML sem lidar com qualquer código de linguagem de programação ou outras camadas do modelo.

Cruz (2007) apresenta um framework genérico e extensível, baseado em uma arquitetura modular e utiliza o conceito de plugins pré-instalados; esta abordagem é bastante flexível, permitindo adicionar, remover ou atualizar serviços sem a necessidade de recompilar o núcleo da plataforma ou de interromper os serviços. Isso garante um maior compartilhamento de código e facilidade de manutenção. A ideia é poder acoplar diferentes tipos de serviços ao framework.

A arquitetura apresentada diferencia-se da proposta de Cruz (2007), de modo que o trabalho apresentado por ela não contempla a customização dos módulos, apenas a estrutura para que sejam adicionados e removidos. No trabalho apresentado por Gunathunga et al. (2007) é proposta a geração de interfaces a partir de arquivos XML, dissociando interface de usuário das regras de negócios, entretando, não aborda o problema de manter customizações, apenas lida com alterações na interface sem impactar a regra de negócio. Deste modo, o presente trabalho busca combinar as duas abordagems para oferecer uma solução viável ao problema de manter várias versões de um mesmo software ao mesmo tempo em que são desenvolvidas customizações para diferentes organizações.

4 ESTUDO DE CASO: APLICAÇÃO DO MECANISMO NA CUSTOMIZAÇÃO DE UM ERP

Neste capítulo é descrita uma aplicação do mecanismo proposto para customizar um ERP. O software analisado neste trabalho está sendo desenvolvido por uma pequena empresa de Londrina-PR há pouco menos de três anos, que é consideravelmente pouco em se tratando de sistemas deste tipo. Ele foi concebido para que pudesse funcionar em diversas plataformas, e havia exigências de que pudesse ser acessado pela Internet; logo, a escolha da plataforma Web foi natural. O sistema compreende funções para atender às principais necessidades de uma organização. Os módulos disponíveis até o momento apoiam a distribuição e expedição, vendas, compras, finanças, contabilidade, gestão de recursos humanos, controle de estoque, engenharia de produtos, gestão da produção, relacionamento com o cliente e frente de caixa.

O sistema é voltado para a indústria de transformação, principalmente moveleira e metalúrgica, sendo que atualmente encontra-se implantado em três nichos diferentes: duas empresas da indústria moveleira, uma empresa fabricante de máquinas e prestação de serviços de laminação de roscas e uma empresa de representações comerciais. Apesar do pouco tempo no mercado, o produto vem recebendo críticas positivas por parte dos usuários, principalmente relacionadas à facilidade de uso.

O software foi construído com o framework Ruby on Rails, que apresenta um ambiente completo para o desenvolvimento de aplicações web. Para Carneiro Júnior et al. (2010), o Rails é um framework completo para construir aplicações web. Possui uma filosofia de código limpo e produtividade. Originalmente criado por David Heinemeier Hansson, a primeira versão do framework foi lançada em julho de 2004, e vem amadurecendo desde então. Construído com a linguagem Ruby, é multiplataforma e possui código aberto e uma comunidade participativa. É uma linguagem interpretada, orientada a objetos, com sintaxe elegante, concisa e expressiva, com um suporte excelente à reflexão, permitindo metaprogramação e sendo bastante adequada para criação de linguagens específicas de domínio (Domain Specific Languages – DSL).

O Ruby on Rails incentiva o conceito de “convenção em vez de configuração”, que significa que só deve ser necessário configurar o que for incomum. Permite iniciar o desenvolvimento reduzindo algumas decisões que devem ser tomadas no início do projeto. Como resultado, é possível focar na resolução do problema e terminar o trabalho rapidamente. O framework apresenta poucos arquivos de configuração, ao contrário

de muitos frameworks atuais, que muitas vezes demandam uma grande carga de configuração antes de serem usados. Em vez de exigir configuração, o framework assume certas convenções de estruturas e nomenclaturas, que se comportam de maneira previsível de acordo com seu significado (CARNEIRO JÚNIOR et al, 2010).

O código do framework Ruby on Rails foi estudado e alterado para suportar o mecanismo proposto. O processo de inicialização foi modificado para que a estrutura dos módulos fosse reconhecida, de forma que as definições dos modelos pudessem ser aplicadas por meio do suporte reflexivo da linguagem Ruby, e os módulos fossem carregados respeitando a precedência das customizações. Também foi adicionado suporte à exibição de interfaces gráficas definidas com o modelo proposto, de forma a coexistir com o código já existente e permitindo a migração gradual. Desta maneira, será possível aplicar os conceitos propostos e verificar sua efetividade sem impactar no restante do software.

Na aplicação da proposta, optou-se por utilizar o sistema Rubygems para empacotamento dos módulos, por ser um formato de distribuição de pacotes bastante popular no macrossistema Ruby. Permite distribuir e instalar códigos em qualquer plataforma onde o Ruby estiver disponível. Trata-se de um sistema de gerenciamento de pacotes para aplicações e bibliotecas que permite resolver automaticamente dependências entre pacotes, levando em consideração versões específicas (BERUBE, 2007).

As gems são essencialmente arquivos compactados que fornecem uma maneira padronizada de descrever uma biblioteca Ruby e seus requisitos por meio de um arquivo especial chamado gemspec. Este arquivo fornece uma visão geral do software, incluindo nome, versão, descrição, quem o escreveu, arquivos que contém, suas dependências, em quais plataformas funciona e assim por diante. Esta especificação é um script Ruby que cria um objeto Gem::Specification; o quadro 9 apresenta um exemplo de gemspec com os atributos mais comuns (BERUBE, 2007).

Quadro 9 – Exemplo de gemspec

```
Gem::Specification.new do |spec|
  spec.name = "exemplo"
  spec.version = "1.0.0"
  spec.summary = "Exemplo de gemspec"
  spec.author = "Nome do autor"
  spec.email = "email@dominio"
  spec.homepage = "http://www.exemplo.com"
  spec.add_dependency("gem1", ">1.0.0")
  spec.add_dependency("gem2", "~>2.3.0")
  spec.files = FileList['*']
end
```

Fonte: Berube (2007, p. 75)

Este arquivo gemspec será utilizado como fonte de metadados sobre o módulo na aplicação desta proposta. Para os scripts de instalação e alteração do banco, foi utilizado o conceito de migrações do Ruby on Rails que, segundo Carneiro Júnior et al. (2010), é uma forma conveniente de alterar o banco de dados, que possibilita descrever transformações usando a linguagem Ruby em vez de SQL (Structured Query Language), permitindo criar, remover e alterar tabelas de maneira independente de banco de dados. Além disso, o framework também rastreia quais migrações foram executadas, evitando que alterações sejam efetuadas mais de uma vez, e mantendo um histórico da evolução da estrutura, simplificando o versionamento. Um exemplo de migração é apresentado no quadro 10.

Quadro 10 – Exemplo de migração

```
class CreateProducts < ActiveRecord::Migration
  def self.up
     create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
     end
  end
  def self.down
     drop_table :products
  end
end
```

Fonte: Carneiro Jr et. al. (2010, p. 52)

Neste estudo de caso, foram identificadas três características necessárias para que o módulo fosse integrado ao núcleo. Constatou-se que havia necessidade de registrar as interfaces que o módulo implementa, informar as permissões exigidas para cada uma de suas funcionalidades, bem como adicioná-las ao menu do sistema. Assim, foram criadas APIs para estas operações. Para definir as funcionalidades e suas permissões, pode ser utilizada a seguinte chamada no manifesto:

```
Permission.require(level, controller, action)
```

São exigidos três parâmetros para fazer o registro de uma funcionalidade. O primeiro é o nível de acesso necessário para acessá-la, pode ser especificado como :read, :write, :update, e :delete. O segundo e terceiro indicam o controlador e ação que estão sendo protegidos pela permissão. Para registrar uma entrada no menu do sistema, usa-se a chamada:

```
Menu.add(module_name, context, label, url)
```

O parâmetro module_name especifica o macromódulo a que a funcionalidade está relacionada. O macromódulo representa um grande conjunto de funcionalidades relacionadas ligadas a uma vasta área do negócio, como comercial, financeiro, contabilidade etc. Context representa o tipo de funcionalidade, podendo ser :cadastros, :operacional e :relatorios; label é o rótulo que será apresentado ao usuário, e finalmente a URL para acessar o recurso. As entradas no menu serão agrupadas por módulo e contexto, e finalmente serão ordenadas alfabeticamente pelo rótulo que será apresentado ao usuário.

Além dessas características, um módulo pode registrar quais interfaces implementa, para que sejam utilizadas por outros. As dependências devem ser registradas da seguinte forma:

```
dep = DependencyManager.instance
dep.register(:logger) { Logger.new }
```

O código responsável pela instanciação é provido na forma de bloco na linguagem Ruby, armazenado e executado apenas quando necessário. Blocos são como funções de alta ordem que podem ser passadas para métodos e terem a execução deferida. Os outros módulos podem pedir pelas dependências da seguinte forma:

```
dep = DependencyManager.instance
dep.logger
```

O gerenciador de dependências se encarrega de instanciar o objeto de acordo com o registro. Caso sejam necessários parâmetros, é possível informá-los ao registrar a dependência:

```
dep.register(:logger) { Logger.new("logfile.log") }
```

Se for desejável permitir que o parâmetro seja especificado, é possível utilizar:

```
dep.register(:logger) { |c| Logger.new(c.log_filename) }
```

E depois especificar o parâmetro:

```
dep.register(:log_filename) { "logfile.log" }
```

Além da injeção por construtor, também é possível especificar dependências utilizando métodos de acesso de um objeto. A seguir é mostrado um exemplo das possibilidades.

```
dep.register(:objeto) { |c|
   obj = Objeto.new(c.nome)
   obj.logger = c.logger
   obj
}
```

Existe a possibilidade de que uma dependência seja registrada com uma prioridade mais baixa, que pode ser especificada como segue:

```
dep.register(:logger, :priority => :low) { SimpleLogger.new }
```

Este tipo de registro é útil caso um módulo possua uma dependência opcional e tenha uma implementação simples da interface apenas para funcionar isoladamente. Quando um módulo que fornece uma implementação mais elaborada for instalado, ela será utilizada. Além disso, se um módulo customizado fornecer a mesma interface, sua precedência será maior, e ela será utilizada.

Para demonstrar como a customização é feita, será apresentado um exemplo baseado em um cadastro simples de pedidos. Tentou-se formular um exemplo simples o bastante para ser entendido, mas visando demonstrar o maior número de recursos possível. Serão adicionados vários campos ao cadastro original, cuja estrutura de arquivos é detalhada nos quadros a seguir.

Quadro 11 – Arquivo modules/pedidos/db/migrations/create_pedidos.rb

```
class CreatePedidos <
ActiveRecord::Migration
  def self.up
    create_table :pedidos do |t|
      t.integer :cliente_id, :null => false
      t.date :data, :null => false
      t.integer :comissao
    end

    create_table :pedido_produtos do |t|
      t.integer :pedido_id, :null => false
      t.integer :produto_id, :null => false
      t.integer :qtde
    end

    create_table :produtos do |t|
      t.string :nome, :null => false
      t.decimal :preco
    end
  end

  def self.down
    drop_table :pedidos
    drop_table :pedido_produtos
    drop_table :produtos
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 12 – Arquivo modules/pedidos/app/models/pedidos.rb.def

```
model: Pedido
  fields:
  - name: cliente_id
    label: “Cliente”
    type: integer
    validations:
      required: true
  - name: data
    label: “Data”
    type: date
    validations:
      required: true
  - name: comissao
    label: “Comissão %”
    type: integer
```

```
  default_value: 0
  validations:
    custom: ComissaoValidator
associations:
- name: cliente
  label: “Cliente”
  type: belongs_to
  foreign_key: cliente_id
  class_name: Pessoa
- name: pedido_produtos
  type: has_many
  foreign_key: pedido_id
  class_name: PedidoProduto
- name: produtos
  type: has_many
  foreign_key: produto_id
  class_name: Produto
  through: pedido_produtos
validates_with: PedidoValidator
```

Fonte: Desenvolvimento próprio, 2011

Quadro 13 – Arquivo modules/pedidos/app/models/pedido_produto.rb.def

```
model: PedidoProduto
  fields:
  - name: qtde
    type: integer
  - name: produto_id
    type: integer
  association:
  - name: produto
    type: belongs_to
    foreign_key: produto_id
    class_name: Produto
```

Fonte: Desenvolvimento próprio, 2011

Quadro 14 – Arquivo modules/pedidos/app/models/produto.rb.def

```
model: Produto
  fields:
  - name: nome
    type: string
  - name: preco
    type: decimal
```

Fonte: Desenvolvimento próprio, 2011

Quadro 15 – Arquivo modules/pedidos/app/models/pedido.rb

```
class Pedido < ActiveRecord::Base
  def preco_total
    pedido_produtos.map {|p| p.qtde * p.produto.preco}.sum
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 16 – Arquivo modules/pedidos/app/models/pedido_validator.rb

```
class PedidoValidator < ActiveModel::Validator
  def validate
    if record.pedido_produtos.empty?
      record.errors[:base] << 'Devem ser adicionados produtos ao
pedido'
    end
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 17 – Arquivo modules/pedidos/app/models/comissao_validator.rb

```
class ComissaoValidator < ActiveModel::EachValidator
  def validate_each(object, attribute, value)
    if value > 20
      object.errors[attribute] << "comissão excede valor máximo"
    end
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 18 – Arquivo modules/pedidos/app/models/workflow_pedido.rb

```
class WorkflowPedido
  def run(params)
    pedido = Pedido.create(params[:pedido])
    deps = DependencyManager.instance
    regra_comissoes = deps.regra_comissoes
    valor_comissao = regra_comissoes.calcular(pedido)
    # … outras instruções ...
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 19 – Arquivo modules/pedidos/app/models/regra_comissoes.rb

```
class RegraComissoes
  def calcular(pedido)
    pedido.preco_total * (pedido.comissao /
100)
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 20 – Arquivo modules/pedidos/app/controllers/pedidos_controller.rb

```
class PedidosController
  # … outros métodos ...

  def create
    deps = DependencyManager.instance
    workflow_pedido = deps.workflow_pedido
    workflow_pedido.run(params)
  end

  # … outros métodos ...
End
```

Fonte: Desenvolvimento próprio, 2011

Quadro 21 – Manifesto do módulo

```
Permission.require_crud(“pedidos”)
Menu.add(“Comercial”, :cadastros, “Pedidos”,
“/pedidos/index”)

deps = DependencyManager.instance
deps.register(:workflow_pedido) { WorkflowPedido.new }
deps.register(:regra_comissoes) { RegraComissoes.new }
```

Fonte: Desenvolvimento próprio, 2011

Quadro 22 – Arquivo modules/pedidos/app/views/pedidos/new.html.def

```
title: Novo Pedido
back: “/pedidos/index”

filtered_table:
  url: “/produtos/index”
  per_page: 4
  search_form:
    fields:
    - name: nome
    filter_button_label: Buscar
    clear_button_label: Limpar
  columns:
  - name: id
    label: “Código”
  - name: nome
  - name: add
    label: Adicionar
    sortable: false
    convert_with: AddLinkGenerator

form:
  action: “/pedidos/create”
  fields:
  - name: cliente
    dinamic_collection:
      data_url: “/pessoas/index”
      param_name: nome
      colums:
      - name: nome
  - name: data
    mask: ##/##/####
    - name: pedido_produtos
      subfields:
      - name: qtde
      - name: produto_id
        hidden: true
      - name: “produto#nome”
        readonly: true
      - name: “produto#preco”
        readonly: true
    - name: comissao
      label: “Comissão %”
    button:
      name: salvar
      label: Salvar
```

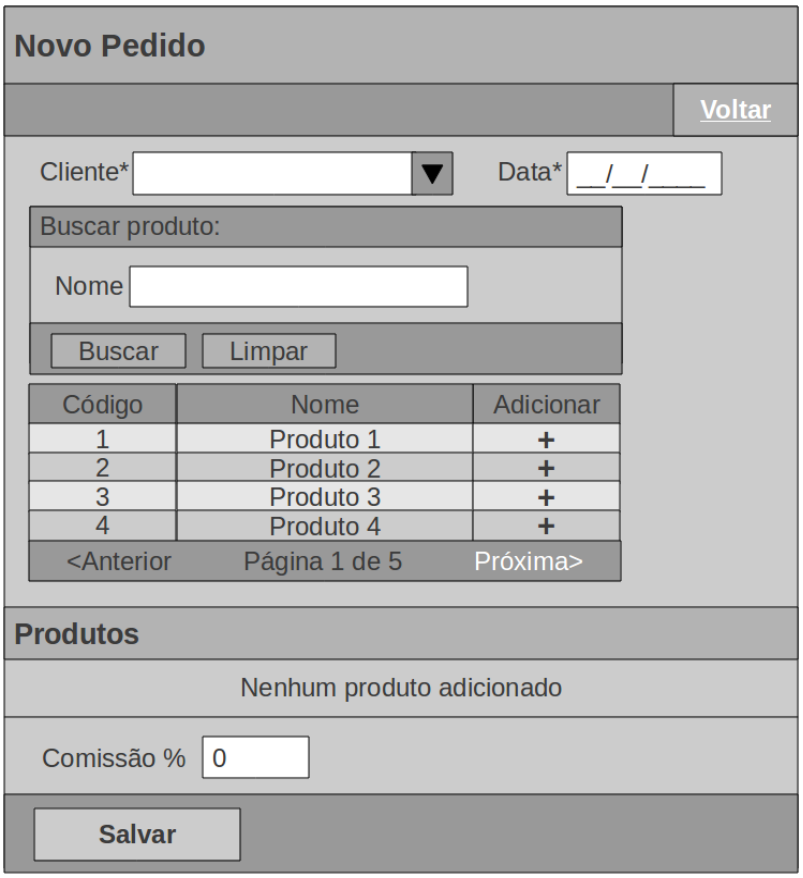
Fonte: Desenvolvimento próprio, 2011

Quadro 23 – Arquivo modules/pedidos/app/models/add_link_generator.rb

```
class AddLinkGenerator
  def self.as_string(record)
    link_to_function(“+”,
“adicionarProduto(#{record.to_json)”)
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Figura 22 – Interface gerada pelas definições



Fonte: Desenvolvimento próprio

A definição subfields cria um template JavaScript, que será populado para que seja adicionado ao formulário com os valores do produto selecionado.

Quadro 24 – Template gerado pela declaração subfields

```
<script type="text/x-jquery-template" title="pedido_produto">
  <div class=”pedido_produto”>
    <label>Qtde</label>
    <input name=”pedido[pedido_produtos][qtde]” value=”<%= pedido_produto.qtde %>”
type=”text”>

    <input name=”pedido[pedido_produtos][produto_id]” value=”<%=
pedido_produto.produto_id %>”
           type=”hidden”>

    <label>Produto</label>
    <input name=”pedido[pedido_produtos][produto_nome]” value=”<%=
pedido_produto.produto.nome %>”
          type=”text” readonly=”readonly”>
    <input name=”pedido[pedido_produtos][produto_preco]” value=”<%=
pedido_produto.produto.preco %>”
          type=”text” readonly=”readonly”>
    <a href=”#” onclick=”$(this).parents('.pedido_produto').remove()”>x</a>
  <div>
</script>
```

Fonte: Desenvolvimento próprio, 2011

É possível ver os campos para entrada de dados, bem como o campo oculto, e o link para remover o produto adicionado. No quadro 25 é mostrada a função Javascript responsável por adicionar um novo produto ao pedido.

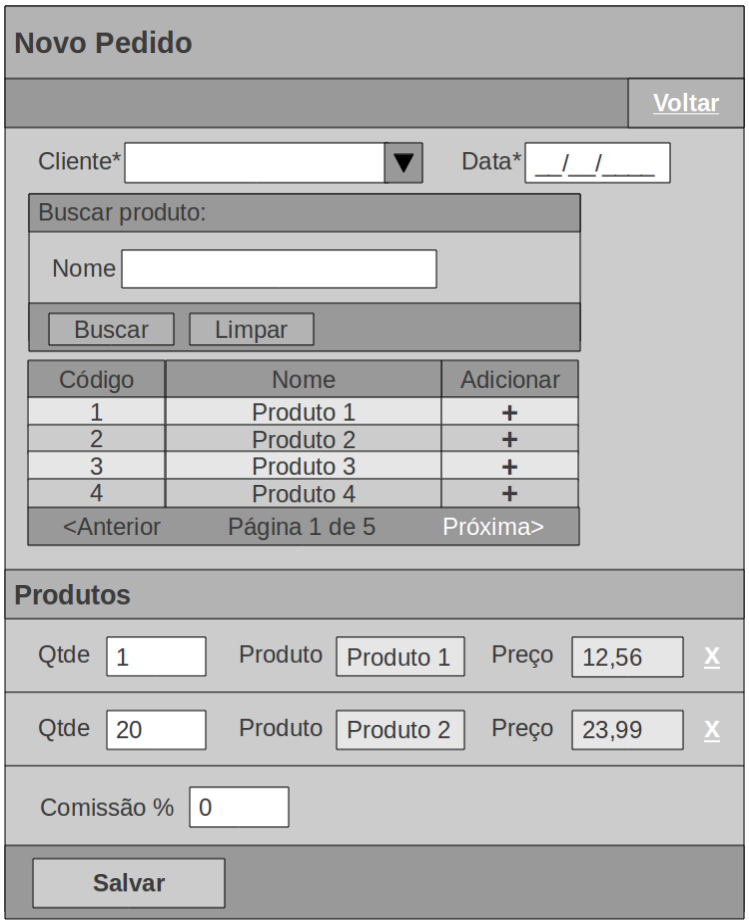
Quadro 25 – Função JavaScript responsável por adicionar um produto ao pedido

```
function adicionarProduto(produto) {
    var html = $.templates.pedido_produto(produto)
    $('#pedido_produtos').append(html);}
```

Fonte: Desenvolvimento próprio, 2011

O funcionamento é relativamente simples, ela recebe o produto como JSON (JavaScript Object Notation), popula o template com os dados, e adiciona o resultado do template à área reservada aos produtos. A figura 23 ilustra a aparência dos campos adicionados dinamicamente.

Figura 23 – Exemplo de adição dinâmica de campos



Fonte: Desenvolvimento próprio

Após apresentar as definições originais, são apresentados os arquivos responsáveis por fazer a customização. Em primeiro lugar, são feitas as alterações no banco durante a instalação da customização.

Quadro 26 – Arquivo customizations/pedidos/db/migrations/change_pedidos.rb

```
class  ChangePedidos < ActiveRecord::Migration
  def self.up
    remove_column :pedidos, :comissao
    add_column :pedido_produtos, :comissao, :integer,
:default => 0

    create_table :pedido_telefones do |t|
      t.integer :pedido_id, :null => false
      t.string :numero, :null => false
      t.integer :tipo
    end
  end

  def self.down
    drop_table :pedido_telefones
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 27 – Arquivo customizations/pedidos/app/models/pedido.rb.def

```
model: Pedido
  fields:
  - name: comissao
    remove: true
  associations:
  - name: telefones
    type: has_many
    foreign_key: pedido_id
    class_name: Telefone
  validates_with:
PedidoCustomizadoValidator
```

Fonte: Desenvolvimento próprio, 2011

Quadro 28 – Arquivo customizations/pedidos/app/models/pedido_produto.rb.def

```
model: PedidoProduto
  fields:
  - name: comissao
    type: integer
    default_value: 0
```

Fonte: Desenvolvimento próprio, 2011

Quadro 29 – Arquivocustomizations/pedidos/app/models/pedido_customizado_validator.rb.def

```
class PedidoCustomizadoValidator < ActiveModel::Validator
  def validate
    if record.pedido_produtos.empty?
      record.errors[:base] << 'Devem ser adicionados produtos ao pedido'
    elsif record.pedido_produtos.any? {|p| p.comissao > 20 }
      record.errors[:base] << 'Comissão excede valor máximo'
    end
  end
end
```

Fonte: Desenvolvimento próprio, 2011

Quadro 30 – Arquivo customizations/pedidos/app/views/pedidos/new.html.def

```
form:
  fields:
    - name: telefones:
      subfields:
      - name: numero
      - name: tipo
      insert_after: data
      collection:
        key_class: integer
        value_class: string
        entries: [ [1, “Residencial”], [2, “Comercial”] ]
    - name: pedido_produtos
      subfields:
      - name: comissao
    - name: comissao
      remove: true
```

Fonte: Desenvolvimento próprio, 2011

Quadro 31 – Arquivo customizations/pedidos/app/models/regra_comissoes_customizada.rb

```
class RegraComissoesCustomizada
  def calcular(pedido)
    pedido.pedido_produtos.map {|p| p.qtde * p.produto.preco  * (p.comissao / 100)}.sum
  end
end
```

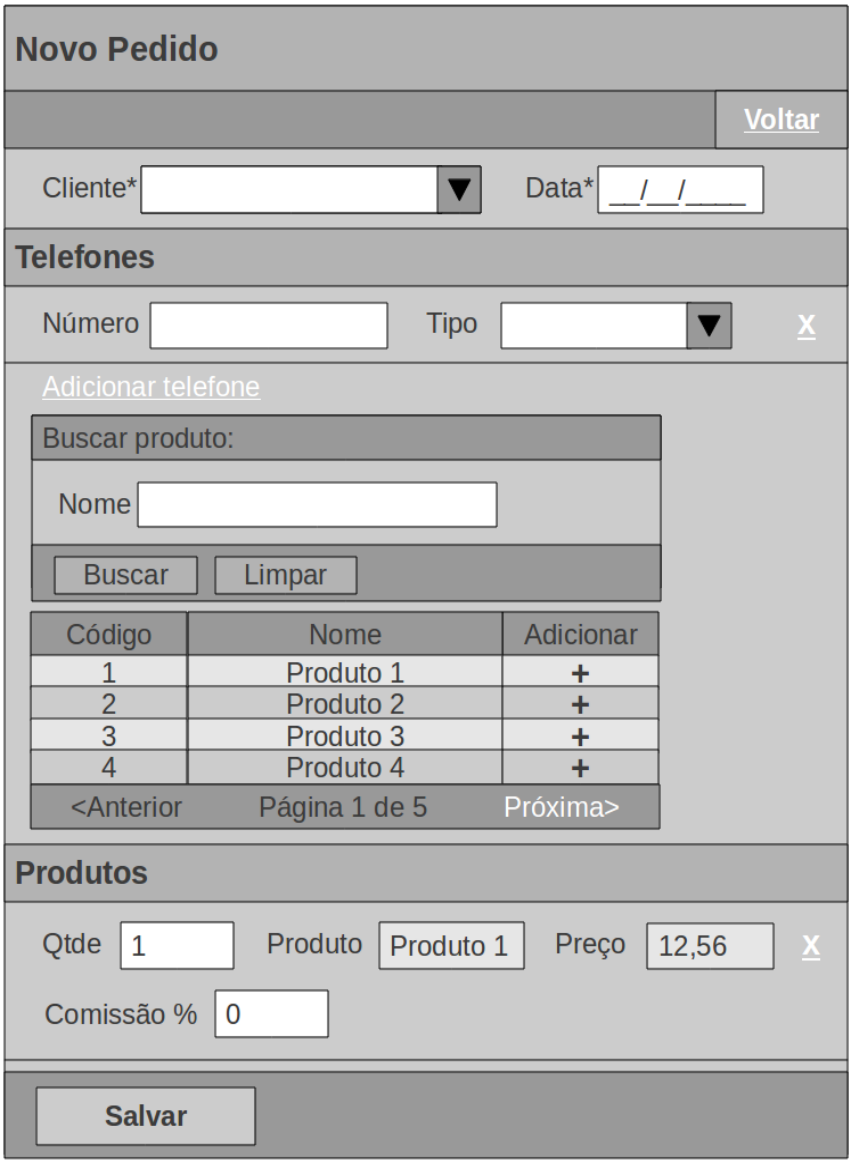
Fonte: Desenvolvimento próprio, 2011

Quadro 32 – Manifesto do módulo customizado

```
deps.register(:regra_comissoes) {
RegraComissoesCustomizada.new }
```

Fonte: Desenvolvimento próprio, 2011

Figura 24 – Resultado das customizações aplicadas



Fonte: Desenvolvimento próprio, 2011

Foi demonstrado como é possível fazer customizações. Em casos extremos, se a regra de negócio for totalmente diferente do sistema, existe a possibilidade de substituir o módulo, mantendo sua interface.

4.1 VIABILIDADE DA PROPOSTA

Para mensurar a efetividade da proposta foi feita uma comparação de vários tipos de customizações (conforme Tabela 21) usando a metodologia anterior, que levava à proliferação de versões, e a arquitetura proposta. Foi usado como base de comparação um programador com três anos de experiência no desenvolvimento do ERP, e se tomou-se como hipótese outros três programadores com diferentes experiências, descritas na tabela 20, de forma que sua capacidade foi estimada.

Tabela 20 – Experiência considerada dos programadores

Programador	Experiência
Programador 1	3 anos
Programador 2	2 anos e 3 meses
Programador 3	1 ano e 6 meses
Programador 4	1 ano e 2 meses

Fonte: Desenvolvimento próprio, 2011

O programador executou todas as cinco alterações usando as dois métodos, e os tempos gastos em cada customização foram medidos em minutos. As customizações que foram utilizadas na comparação estão descritas na tabela 21, em ordem crescente de complexidade. Com base na experiência considerada para os outros programadores, foi feita uma projeção do tempo que levariam para executar as mesmas tarefas, estando os resultados expostos na tabela 22. A partir desses dados, foi calculado o ganho médio de tempo que cada programador poderia obter, caso a projeção se confirme. Estes resultados estão relacionados na tabela 23.

Tabela 21 – Tipos de customização comparadas

Customização	Descrição
Customização 1	Remover um campo
Customização 2	Adicionar um campo do tipo string
Customização 3	Alteração de relatório adicionando dois atributos e removendo um
Customização 4	Adicionar três campos, sendo: • um do tipo data • os outros dois compõem um relacionamento do tipo 1:N, e possuem o tipo inteiro
Customização 5	Adicionar sete campos, sendo: • um do tipo inteiro, com máscara e conversão personalizada • um do tipo string • um relacionamento do tipo 1:1 • os quatro restantes compõem uma relação N:M, de forma que um é do tipo string, dois do tipo inteiro e um do tipo data

Fonte: Desenvolvimento próprio, 2011

Tabela 22 – Comparação do tempo (em minutos) de customização da metodologia anterior com a proposta.

Customização	Programador 1		Programador 2		Programador 3		Programador 4	
	Anterior	Proposta	Anterior	Proposta	Anterior	Proposta	Anterior	Proposta
1	15	6	19	9	22	14	24	15
2	30	18	34	20	39	26	39	33
3	46	22	51	23	62	31	65	40
4	97	25	108	30	117	42	121	44
5	203	50	242	58	268	69	270	71

Fonte: Desenvolvimento próprio, 2011

Tabela 23 – Ganho médio de tempo ao usar a arquitetura proposta

	Programador 1	Programador 2	Programador 3	Programador 4
Ganho Médio	33%	33,83%	40,32%	45,21%

Fonte: Desenvolvimento próprio, 2011

Como pode ser visualizado na tabela 23, de modo geral, a proposta, se confirmada, poderia permitir que os programadores executassem mais rapidamente as customizações, contribuindo para uma redução considerável do tempo necessário para a customização. É importante observar que os tempos projetados também são influenciados pelo nível extra de abstração proporcionado pelos modelos, reduzindo a quantidade de código, e consequentemente os erros de implementação.

4.2 Análise dos Resultados

Os resultados apresentados podem servir de indicativos para a utilização da proposta formulada, a qual acena para um possível aumento de produtividade no desenvolvimento de customizações em interfaces gráficas. Foi observado que a proposta, além de interferir menos no gerenciamento de versões do software, mantendo customizações isoladas, pressupõe que caso a hipótese seja verificada, haveria uma ganho médio de 38% de tempo em relação à metodologia anterior. O modelo desenvolvido traz ainda como vantagem a padronização das telas de todo o sistema, fazendo com que mantenham sempre a mesma aparência, espaçamento, cores, tamanho etc, simplificando o desenvolvimento e facilitando futuras alterações nos aspectos visuais da interface.

Como desvantagens, pode-se destacar a curva de aprendizado relacionada à necessidade de aprender a estrutura dos módulos, e também os modelos de descrição. Além disso, também existe uma sobrecarga ocasionada pelo processo de geração de interfaces, que, apesar de imperceptível ao usuário final, deve ser levada em consideração pois pode impactar o desempenho da aplicação em situações de alta concorrência. Em casos em que seja experimentada degradação do desempenho, pode-se usar técnicas de caching para evitar a geração de interfaces repetidamente.

5 CONCLUSÃO

Gestão de sistemas na presença de requisitos voláteis, mutantes e altamente dinâmicos pode ser uma tarefa desafiadora. Projetar para facilitar a mudança é uma tarefa muito mais complexa que simplesmente satisfazer requisitos fixos e bem definidos. Entretanto, novas exigências são inerentes a muitos sistemas, principalmente ERPs, dessa forma, é necessário planejar visando mudanças. Como visto, customização é uma tarefa complexa, custosa e que demanda grande esforço e vasta quantidade de tempo, afigurando-se um sério problema.

A partir da necessidade de minimizar os esforços com customizações, foram estudadas algumas ferramentas para simplificar essa tarefa, visando reduzir o tempo de desenvolvimento, o custo e os defeitos ocasionados pelas customizações. Com base na pesquisa bibliográfica, observou-se que a modularidade apresenta vantagens que podem ser aproveitadas para customização de software, ao permitir acréscimo ou alteração de funcionalidades. Combinada com técnicas de geração de código e reflexão computacional, obtém-se um conjunto consistente para o processo de customização.

Neste trabalho foi apresentada uma arquitetura cujos resultados têm se mostrado promissores no que diz respeito à customização de software. Na proposta que se formulou, identifica-se que a abstração em conjunto com a injeção de dependências, um ambiente dinâmico como a web, e bom suporte à reflexão foram fundamentais para o desenvolvimento da proposta, tendo em vista a necessidade de geração dinâmica de código a partir da combinação de modelos por meio de regras de precedência.

Obteve-se uma indicação da sua efetividade por meio da comparação com a antiga abordagem. Os problemas relacionados à proliferação de versões foram reduzidos, e o objetivo de manter o código customizado isolado do original foi alcançado, de forma a causar menos impacto na gerência de configuração do projeto. Apesar da curva de aprendizado relacionada à necessidade de aprender a estrutura dos módulos, e também os modelos de descrição, com base nos resultados apresentados é possível inferir que a arquitetura proposta pode oferecer maior produtividade para o desenvolvimento das customizações mais comuns.

O grau de abstração adicionado pelos modelos propostos permitiu uma redução considerável no tempo e os custos necessários para as customizações mais comuns, aumentando a produtividade da equipe, permitindo um ganho médio de 38% de tempo em relação à metodologia anterior. O modelo desenvolvido traz ainda como vantagem a padronização das telas de todo o sistema, fazendo com que mantenham sempre a mesma

aparência, simplificando o desenvolvimento e facilitando a alteração dos aspectos visuais da interface.

De forma geral, a proposta apresentada pode ser utilizada para o desenvolvimento de qualquer sistema web que venha a necessitar de customizações, não apenas ERPs. Entretanto, embora a estrutura apresentada venha atendendo as necessidades de customização satisfatoriamente, é possível que surjam requisitos que não possam ser atendidos com as ferramentas demonstradas. Os modelos apresentados podem ser complementados futuramente para suportar novos requisitos.

5.1 CONTRIBUIÇÕES

De forma geral, a proposta apresentada pode ser utilizada para o desenvolvimento de qualquer sistema web que venha a necessitar de customizações nas interfaces gráficas, não apenas ERPs. Além disso, este trabalho apresenta outras duas contribuições, decorrentes das características exploradas para lidar com as customizações. A primeira está relacionada à arquitetura proposta, que pode ser utilizada como uma referência para construção de arquiteturas modulares, servindo como exemplo de implementação e dos elementos necessários em uma arquitetura deste tipo. A segunda diz respeito aos modelos que foram criados, que podem ser utilizados isoladamente para o rápido desenvolvimento de interfaces.

As respostas encontradas podem ser válidas em outras situações nas quais seja necessário promover customizações. Entretanto, é oportuno enfatizar que a presente proposta não visa fornecer respostas definitivas em relação à customização de sistemas, ou mesmo arquiteturas modulares. Entende-se que a execução deste trabalho representa uma constatação da efetividade das técnicas utilizadas e assume o papel de fomentar a pesquisa de novas alternativas para os problemas apresentados. Espera-se, assim, que a proposta formulada possa servir de substrato para ao desenvolvimento de novas propostas que venham a complementar as ideias que serviram de base para a formulação deste trabalho.

5.2 TRABALHOS FUTUROS

Visando contornar os problemas apresentados, a proposta objetivou cobrir os casos mais comuns de customização que foram observados, relacionados às interfaces com o usuário, e, portanto, não fornece uma resposta definitiva para todas as necessidades de

customização. É provável que conforme o software seja implantado em diferentes segmentos, a proposta precise de revisão e evolução para suportar novos requisitos.

Desta forma, sugere-se para futuras pesquisas o estudo de customizações mais complexas, envolvendo interfaces muito dinâmicas, e também customizações agressivas nos processos do software.

REFERÊNCIAS

ARINZE, Bay; ANANDARAJAN, Murugan. A Framework for Using OO Mapping Methods to rapidly configure ERP systems. Communications of the ACM. v. 46, n. 2. fev. 2003.

BARCELOS, R. Avaliando Modelos Arquiteturais através de um checklist baseado em atributos de qualidade. In: MÜLLER, Alex Moreira. Papel de uma arquitetura modular no desenvolvimento distribuído de software: um estudo de caso no Yart. 2006. Monografia (Pós-Graduação Lato Sensu) – Universidade Federal de Lavras, Minas Gerais, 2006.

BARSTOW, David; ARANGO, Guillermo. Designing software for customization and evolution. Proceedings of the Sixth International Workshop on Software Specification and Design. Como, Itália. out, 1991.

BENKLER, Y. Coases Penguin, or, Linux and The Nature of the Firm. In: MÜLLER, Alex Moreira. Papel de uma arquitetura modular no desenvolvimento distribuído de software: um estudo de caso no Yart. 2006. Monografia (Pós-Graduação Lato Sensu) – Universidade Federal de Lavras, Minas Gerais, 2006.

BERUBE, David. Practical Ruby Gems. Apress. 2007.

BOSWELL, David; KING, Brian; OESCHGER, Ian; COLLINS, Pete; MURPHY, Eric. Creating Applications with Mozilla. O'Reilly, set., 2002.

BOZZON, Alessandro, COMAI, Sara; FRATERNALI, Piero; CARUGHI, Giovanni Toffetti. Conceptual Modeling and Code Generation for Rich Internet Applications. ICWE’06, July 11-14, 2006, Palo Alto, California, USA.

CARNEIRO JUNIOR, Cloves. HARDY, Jeffrey Allan; CATLIN, Hampton. Beginning Rails: From Novice to Professional. Apress, 2007.

CICCHETTI, Antonio; DI RUSCIO, Davide; DI SALLE, Amleto. Software Customization in Model Driven Development of Web Applications. Dipartimento di Informatica Universit `a degli Studi dell’Aquila.L’Aquila, Italy. Março, 2007.

COELHO, Frederico de Miranda. Uso de componentes de software no desenvolvimento de Frameworks orientados a objetos. 2002. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Campinas, Campinas, 2002.

CORRÊA, Sand Luz. Implementação de sistemas tolerantes a falhas usando programação reflexiva orientada a objetos. 1997. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Campinas, Campinas, 1997.

CRUZ, Ariadne Arrais. Projeto e implementação de um framework para weblabs baseado em ajax e padrões de projeto. 2007. Dissertação de Mestrado (Engenharia Elétrica) – Universidade Estadual de Campinas. Campinas, 2007.

DAVENPORT T. H. Davenport, Putting the enterprise into the enterprise system. Harv. Bus. Rev., pp. 121–131, July-August 1998. In: LUO E STRONG, Wenhong; STRONG, Diane M. A Framework for Evaluating ERP Implementation Choices. Ieee Transactions On Engineering Management, v. 51, n. 3, Agosto, 2004.

DITTRICH, Yvonne; VAUCOULEUR, Sebastien. Practices Around Customization of Standard Systems. CHASE'08. Leipzig, Germany. Maio, 2008.

DOR, Nurit; LEV-AMI, Tal; LITVAK, Shay; SAGIV, Mooly; WEISS, Dror. Customization Change Impact Analysis for ERP Professionals via Program Slicing. ISSTA'08, Seattle, Washington, USA. 2008.

DROMEY, R. Architcture as an Emergent Property of requirements Integration. In: MÜLLER, Alex Moreira. Papel de uma arquitetura modular no desenvolvimento distribuído de software, um estudo de caso no Yart. 2006. Monografia (Pós-Graduação Lato Sensu) – Universidade Federal de Lavras, Minas Gerais, 2006.

FAYAD, Mohamed E.; HAMU, David S.; BRUGALI, David. Enterprise Frameworks Characteristics, Criteria, and Challenges. Communications of the ACM . v. 43, n. 10. out., 2000.

FELDT, Kenneth C. Programming Firefox. O'Reilly Media, 2007.

FIELDING, Roy Thomas. Architectural styles and the design of network-based software architectures. 2000. Tese (Doutorado em Ciência da Computação) – University of California, Irvine. 2000.

FLANAGAN, David. JavaScript: The Definitive Guide, 5 ed. O'Reilly Media 2006

FOOTE, Brian. Designing to facilitate change with object-oriented Frameworks. Dissertação (Mestrado em Ciência da Computação) - University of Illinois at Urbana-Champaign, 1988.

FOOTE, Brian. An object-oriented Framework for reflective metalevel architectures. Tese (Doutorado em Ciência da Computação) - University of Illinois at Urbana-Champaign, 1994.

FOWLER, Martin. Inversion of Control Containers and the Dependency Injection pattern. Disponível em http://martinfowler.com/articles/injection.html. 2004. Acesso em: 14 Nov 2010.

FOWLER, Martin; RICE, David, FOENNEL, Matthew, HIEETT, Edward, MEE, Robert, STAFFORD, Randy. Patterns of Enterprise Application Architecture, Addison Wesley, 2002.

FRATERNALI, Piero; COMAI, Sara BOZZON, Alessandro. Engineering Rich Internet Applications with a Model-Driven Approach. ACM Transactions on the Web, Vol. 4, No. 2, Article 7, 2010.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1 ed. 1994.

GARLAN, David; SHAW, Mary. An Introduction to Software Architecture. School of Computer Science Carnegie Mellon University Pittsburgh, PA. Janeiro, 1994.

GARLAN, David; ALLEN, Robert; OCKERBLOOM, John. Architectural Mismatch: why Reuse Is So Hard. IEEE SOFTWARE. v. 12. Novembro, 1995.

GARLAN, David. Software Architecture: a Roadmap. ACM Press. 2000.

GARLAND, Jeff; ANTHONY, Richard. Large-Scale Software Architecture: A Practical Guide using UML. Ed. John Wiley & Sons LTD. 2003.

GOODMAN, Danny. Dynamic HTML: The Definitive Reference. O'Reilly Media, 1998.

GUNATHUNGA, J.D.S.S.; UMAGILIYA, A.U.; KODITUWAKKU, S.R. Leverage the use of XML in dynamic GUI parsing and database stored procedures. Second International Conference on Industrial and Information Systems. ICIIS 2007, 8 – Sri Lanka, Agosto, 2007.

HAU, Elvis; APARÍCIO, Manuela. Software Internationalization and Localization in Web Based ERP. SIGDOC '08: Proceedings of the 26th annual ACM international conference on Design of communication. Lisboa, Portugal. 2008.

IEEE - The Institute of Eletrical and Eletronics Engineers. SWEBOK: Guide to Software Engineering Body of Knowledge. Disponível em: http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf. Acesso em julho de 2010.

JOHNSON, Ralph E.; FOOTE, Brian. Designing Reusable Classes. Journal of Object-Oriented Programming. v. 1, n. 2, p. 22-35. Junho/Julho, 1988.

LARMAN, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and The Unified Process. Prentice Hall, 2002.

LUO E STRONG, Wenhong; STRONG, Diane M. A Framework for Evaluating ERP Implementation Choices. IEEE Transactions on engineering management, v. 51, n. 3, Agosto, 2004.

MAES, Pattie. Concepts and experiments in computational reflection. OOPSLA'87 Conference proceedings on Object-oriented programming systems, languages and applications, v. 22, ed. 12. ACM Nova York, Estados Unidos, Dezembro, 1987.

MALAN Ruth; BREDEMEYER, Dana. Software Architecture: Central Concerns, Key Decisions. Disponível em: http://www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF. Acesso em: 13 de Agosto de 2010.

MARTIN Robert C.; MARTIN, Micah. Agile Principles, Patterns, and Practices in C#. Prentice Hall; 1 ed. Julho, 2006.

MEYER; Bertrand. Object Oriented Software Construction, 2d. ed., Prentice Hall, 1997.

MÜLLER, Alex Moreira. Papel de uma arquitetura modular no desenvolvimento distribuído de software, um estudo de caso no Yart. 2006. Monografia (Pós-Graduação Lato Sensu) Departamento de Ciência da Computação da Universidade Federal de Lavras. Lavras, Minas Gerais, 2006.

NARDUZZO, Alessandro; ROSSI, Alessandro. Modularity in Action: GNU/Linux and Free/Open Source Software Development Model Unleashed. n. 78. Quaderno DISA. Maio. 2003.

NORDHEIM, Stig; PAIVARINTA, Tero. Customization of Enterprise Content Management Systems: An Exploratory Case Study. Proceedings of the 37th Hawaii International Conference on System Sciences. 2004.

OLIVEIRA, André L. C. Metodologia para desenvolvimento de sistemas de informação através da utilização de módulos autônomos. Porto Alegre. Universidade Federal do Rio Grande do Sul. Dissertação de Mestrado. 1998.

ORAM, Andy; WILSON, Greg. Beautiful Code. O'Reilly Media. 2007.

PARNAS, D.L.. On the Criteria To Be Used in Decomposing Systems into Modules. Carnegie-Mellon University. Association for Computing Machinery Inc. v. 15, n. 12, pp. 1053 – 1058. Dezembro, 1972.

PARNAS, David L. Designing Software for Ease of Extension and Contraction. IEEE Transactions on software engineering, v. 5, n. 2, Março, 1979.

PARNAS David Lorge; CLEMENTS, Paul C.; WEISS, David M. The Modular Structure of Complex Systems. IEEE Transactions on software engineering. v. SE-1 1, n. 3, Março, 1985.

REESE, George. Database Programming with JDBC and Java, 2 ed. O'Reilly Media, 2000.

ROTHENBERG, Marcus A.; SRITE, Mark. An Investigation of Customization in ERP System Implementations. IEEE Transactions on engineering management, v. 56, n. 4, Novembro, 2009.

SCHMITT, Cristopher. CSS Cookbook. 3 ed. O'Reilly Media, 2009.

SILVA, Maurício Samy. Construindo sites com CSS e XHTML: sites controlados por folhas de estilo em cascata. São Paulo: Novatec, 2008.

SUSMAN, G. L., EVERED, R. D. An assessment of the scientific merits of action research. Administrative Sciences Quartely, 1978, 23, p.582-603.

TALEVSKI, Alex; CHANG, Elizabeth; DILLON, Tharam S. Meta Model Driven Framework for the Integration and Extension of Application Components. Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03). 2004.

THEMISTOCLEOUS M. THEMISTOCLEOUS, Z. IRANI, R. O'KEEFE; R. PAUL. ERP problems and application integration issues: An empirical survey. In Hawaii Inter. Conf. on System Sciences, 2001. In: DOR, Nurit; LEV-AMI, Tal; LITVAK, Shay; SAGIV, Mooly; WEISS, Dror. Customization Change Impact Analysis for ERP Professionals via Program Slicing. ISSTA'08, Seattle, Washington, USA. 2008.

TRAVASSOS, Guilherme Horta; MEDEIROS, Paulo Sérgio. Praticando a ciência no dia a dia do desenvolvimento de sistemas. Computação Brasil, Porto Alegre, v. 14, p. 22-23, Outubro, 2010

TULACH, Jaroslav. Practical API Design: Confessions of a Java Framework Architect. Apress. 2008.

VALKOV, Svilen. Innovative concept of open source Enterprise Resource Planning (ERP) system. International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing Gabrovo - CompSysTech'08. University of Economics – Varna, Bulgaria. Junho, 2008.

VAROTO, A. C. Visões em arquitetura de software. In: MÜLLER, Alex Moreira. Papel de uma arquitetura modular no desenvolvimento distribuído de software: um estudo de caso no Yart. 2006. Monografia (Pós-Graduação Lato Sensu) – Universidade Federal de Lavras. Lavras, Minas Gerais, 2006.

YAU, Stephen S.; TAWEPONSOMKIAT, Choksing; HUANG, Dazhi. A Framework for Extensible Component Customization for Component-based Software Development. Computer Science and Engineering. Department Arizona State University. Proceedings of the Sixth International Conference on Quality Software (QSIC'06). 2006.

ZHU, Xiyong; XINGWANG, Zheng. A Template-based Approach for Mass Customization of Service-oriented E-business Applications. ICEC’05, Xi’an, China. Agosto, 2005.