Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

ERICKSEN VIANA SAMPAIO

Vinculação de Dados entre Camadas de Software Utilizando Reflexão Computacional

Belo Horizonte
2011

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Análise de Sistemas

Vinculação de Dados entre Camadas de Software Utilizando Reflexão Computacional

por

ERICKSEN VIANA SAMPAIO

Monografia de Final de Curso

Profa. Mariza Andrade da Silva Bigonha
Orientadora

Belo Horizonte
2011

ERICKSEN VIANA SAMPAIO

Vinculação de Dados entre Camadas de Software Utilizando Reflexão Computacional

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como atividade da disciplina Desenvolvimento de Pesquisa e Projetos em Informática, ministrada pela professora Maria de Lourdes Coelho.

Área de concentração: Programação Modular
Orientador(a): Profa. Mariza Andrade da Silva Bigonha

Belo Horizonte
2011

RESUMO

O objetivo do trabalho é desenvolver um mecanismo capaz de capturar os dados de formulários criados dinamicamente e vincular os mesmos em seus respectivos objetos de negócio, de forma automática e que possa ser reutilizado em outros sistemas.
Este mecanismo será aplicado em um sistema de Gestão Eletrônica de Documentos, onde documentos são enviados e posteriormente analisados por uma equipe de auditores. Esta análise é feita extraindo as informações dos documentos em formulários específicos, para cada tipo de documento, gerados em tempo de execução. No fim da análise, as informações extraídas são processadas e persistidas em um banco de dados, servindo de base para a geração de relatórios gerenciais. Para realizar a tarefa proposta será utilizada a biblioteca de reflexão do .NET (*System.Reflection*) , que permite, entre outras funcionalidades, instanciar objetos de uma classe, identificar os atributos, atribuir valores para os mesmos, e invocar métodos da classe, em tempo de execução.

Palavras-chave: Reflexão computacional; Arcabouço .NET; Orientação a Objetos; Programação Modular.

ABSTRACT

The objective of this paper is develop a mechanism able to capture data from forms dynamically created and automatically bind this data in their respective business objects, that can be reused in other systems.This mechanism will be applied to a system of Electronic Document Management, where documents are sent and subsequently analyzed by a team of auditors. This analysis is done by extracting information from documents in specific forms, to each type of document, generated at runtime. After the analysis, the extracted information is processed and persisted in a database, providing the basis for generating management reports.To accomplish the task proposed, the reflection library of the .NET will be used (*System.Reflection*), which allows, among other features, instantiate objects of class, identify those attributes, assigning values to them, and invoke class methods at runtime.

Keywords: Computational Reflection, Framework .NET, Object Orientation, Modular Programming

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BCL	Base Class Library
BLL	Business Logic Layer
CLR	Common Language Runtime
CLS	Common Language Specification
DAL	Data Access Layer
GED	Gestão Eletrônica de Documentos
JIT	Just in Time
JVM	Java Virtual Machine
MSIL	*Microsoft Intermediate Language*
OCP	Open Closed Principle

LISTA DE LISTAGEM

SUMÁRIO

1 INTRODUÇÃO

Os sistemas de softwares estão se tornando cada vez mais complexos, seja devido à natureza do problema a ser resolvido, as necessidades do cliente, ou até mesmo devido a forma como o software é desenvolvido.

Em relação ao processo de desenvolvimento de software, há formas de gerenciar esta complexidade, por exemplo, criando estruturas coesas e de baixo acoplamento (FERREIRA, 2006, p.68), que possam ser reutilizadas em outras aplicações.

As técnicas de reúso de software, além de garantir um nível de qualidade ao software desenvolvido, é uma aliada para a equipe de desenvolvimento, que, na maioria das vezes, necessita desenvolver softwares em um curto prazo de tempo e ao mesmo tempo garantir a qualidade dos mesmos.

Este trabalho propõe a criação de um mecanismo capaz de simplificar e padronizar a obtenção de dados presentes em um formulário e vincular os mesmos de forma automática em seus respectivos objetos de negócio. Atualmente, o processo de captura de dados de formulários está presente na maioria dos sistemas de softwares, seja ele, web, *desktop* ou *mobile*. Este mecanismo pode ser reutilizado, por exemplo, na obtenção de dados em formulários padronizados, como cadastro de clientes ou usuários, proporcionando um ganho de tempo no desenvolvimento dos sistemas.

Para realizar tal tarefa, será utilizada a biblioteca de reflexão do .NET *System.Reflection* (MICROSOFT, 2011), que permite, entre outras funcionalidades, instanciar objetos de uma classe, descobrir os atributos, atribuir valores para os mesmos, e invocar métodos da classe, em tempo de execução.

Com o objetivo de demonstrar o funcionamento do mecanismo proposto, será desenvolvida uma aplicação web, onde dados preenchidos em formulários dinâmicos, gerados em tempo de execução, serão capturados e carregados nas classes de negócios correspondentes, de forma automática.

O sistema a ser desenvolvido se baseia em uma aplicação de Gestão Eletrônica de Documento (GED), onde documentos são enviados por determinadas empresas pelo sistema e estes são armazenados eletronicamente na base de dados. Mais tarde, estes documentos são analisados por uma equipe de auditores, cuja função é extrair os dados do documento, preenchendo o formulário correspondente ao mesmo.

A finalidade é aplicar o mecanismo proposto para capturar os dados dos formulários dinâmicos e vincular os mesmos em seus respectivos objetos de negócio, responsável por realizar as devidas computações sobre os dados, funcionando como uma ponte entre as camadas de interface e negócios.

Com o uso de reflexão computacional (FORMAN, 2004, p.26) é possível vincular os dados da camada de interface para a camada de negócios, de forma automática, sem conhecer a priori qual documento será analisado e quais campos o formulário irá possuir, sendo tudo conhecido em tempo de execução. Tudo isso porque, como observado, a reflexão computacional têm sido usada para instanciar objetos de uma classe, ler e atribuir valores dos atributos contidos nela invocar um método da mesma.

O desenvolvimento de *software* é um processo que, na maioria das vezes, demanda um bom tempo para ser finalizado. O ciclo de desenvolvimento de projetos passa por várias etapas, como levantamento de requisitos, análise, desenvolvimento e implantação. Dependendo da complexidade, um sistema pode ser entregue em meses ou até anos, e, na maioria das vezes, o prazo para o desenvolvimento da solução proposta é curto.

Este trabalho se justifica por disponibilizar um mecanismo capaz de minimizar o tempo de desenvolvimento necessário para coletar dados de formulários, sejam eles estáticos ou criados dinamicamente, descartando a necessidade de implementar um método de captura de dados para cada formulário específico que for gerado. O trabalho proposto será realizado utilizando apenas um método genérico. Com isso, reduz-se o tempo gasto na etapa de construção do *software*.

1.1 Principais Contribuições

A principal contribuição desse trabalho é a criação de um mecanismo capaz de simplificar e padronizar a captura de dados em formulários e vincular os mesmos em seus respectivos objetos de negócios de forma automática, possibilitando sua reutilização em diversos sistemas.

Como outras contribuições, destacamos:

- aplicação dos conceitos de reflexão estudados em uma aplicação *web*.
- implementação de uma arquitetura em três camadas na aplicação proposta.

1.2 Estrutura da Monografia

Seções 2, 3 e 4 são referentes a fundamentação teórica. Seção 2 apresenta os conceitos de reflexão computacional, seus níveis, tipos e protocolos. Seção 3 descreve o arcabouço .NET e a biblioteca de reflexão do arcabouço (*System.Reflection*), seus métodos e aplicações. Seção 4 mostra os conceitos e aplicações do modelo de camadas no desenvolvimento de software. Seção 5 se destina a aplicação prática do trabalho, onde será demonstrado detalhes do desenvolvimento da aplicação proposta. Seção 6 apresenta as conclusões sobre o trabalho desenvolvido, suas limitações e trabalhos futuros.

2 REFLEXÃO COMPUTACIONAL

A reflexão computacional não é um conceito novo. Ela originou-se na área matemática, e hoje é utilizada em mecanismos de alto nível com o objetivo de adicionar novas características a módulos já existentes (BARTH, 2000, p.4).

A reflexão computacional pode ser definida como a habilidade que um programa em execução tem de examinar a si mesmo e ao seu ambiente de software, e alterar o que ele faz dependendo do que foi encontrado (FORMAN,2004, p. 31). Ou seja, uma linguagem de programação que possua o mecanismo de reflexão, permite a determinado programa avaliar sua estrutura e até alterar o seu próprio comportamento em tempo de execução. Esta avaliação e alteração vai depender das capacidades reflexivas presentes na linguagem de programação utilizada.

Um sistema que usa a reflexão computacional em seu ambiente é capaz de analisar suas próprias ações, e, com base nas informações obtidas, ajustá-las conforme a variedade de condições a que é submetido (GOLM, 1998, p.2).

O mecanismo de reflexão computacional possibilita, entre outras funcionalidades, a identificação de métodos, atributos, construtores, bem como os modificadores de acesso de determinada classe presente em um sistema. A fim de examinar a estrutura das classes, a reflexão permite a construção de navegadores de classes, depuradores e ferramentas de teste (SUN MICROSYSTEMS, 2011).

Quando falamos em analisar a estrutura, estamos nos referindo às informações contidas e manipuláveis em um programa, como, por exemplo, classes, objetos, eventos, métodos e estados.

A reflexão computacional possibilita não somente analisar a estrutura interna de um sistema em tempo de execução, mas também obter instâncias de classes, criar novas estruturas, invocar métodos, e até mesmo alterar o comportamento do mesmo, interceptando as mensagens que são trocadas entre seus elementos e manipulando-as conforme a necessidade (SULLIVAN, 2001, p. 12).

Em relação à implementação dessa técnica, cuidados devem ser tomados. Questões relacionadas a restrições de segurança devem ser avaliadas, uma vez que o uso da reflexão requer permissão de acesso em tempo de execução para analisar a estrutura interna de uma determinada classe (SUN MICROSYSTEM, 2011).

Outro problema com o uso dessa técnica está relacionado ao desempenho, pois acessar diretamente informações relacionadas aos elementos, propriedades, métodos, construtores, etc, presentes em uma classe, é mais rápido do que utilizar a reflexão. Chega ao ponto de, uma invocação de método usando reflexão, poder custar cem vezes mais do que se fosse escrita em tempo de compilação (LOPES, 2011, p.26)

Devido ao exposto, é importante saber quando utilizar a reflexão computacional. O seu uso indiscriminado pode gerar instabilidade no sistema e problemas relacionados ao desempenho da aplicação e segurança dos dados.

2.1 Níveis da Arquitetura Reflexiva

A reflexão computacional é composta de uma arquitetura em dois níveis: o nível base e o meta-nível. O primeiro contém os componentes do sistema, suas classes, objetos e estados. O segundo é uma representação das entidades contidas no nível base, sendo representado por metaclasses e meta-objetos. No meta-nível, estão presentes os metadados referentes ao nível base e, por meio dele, é possível realizar a reflexão computacional.

O processo da reflexão computacional pode-se iniciar tanto por meio do objeto presente no nível base, quanto pelo sistema. Quando iniciado pelo objeto do nível base, é necessário que este possua uma identificação correspondente ao seu objeto no meta-nível. Se a responsabilidade de iniciar processo for do sistema, qualquer operação envolvendo o objeto no nível base irá ativar seu meta-objeto correspondente (BARTH, 2000, p.32).

O uso da reflexão computacional traz um grande benefício: a adaptabilidade. Por meio do meta-nível, é possível monitorar, estender ou mesmo, alterar o funcionamento do nível base, sem que este tenha que ser re-implementado (YAMAGUTI, 2002, p. 62).

A Figura 1, extraída de Souza, (SOUZA, 2001, p. 292), mostra a representação desses dois níveis. Como pode ser visto, um objeto no meta-nível pode estar associado a um ou mais objetos no nível base. Da mesma forma, podemos ter cada objeto do meta-nível associado a seu respectivo objeto no nível base.

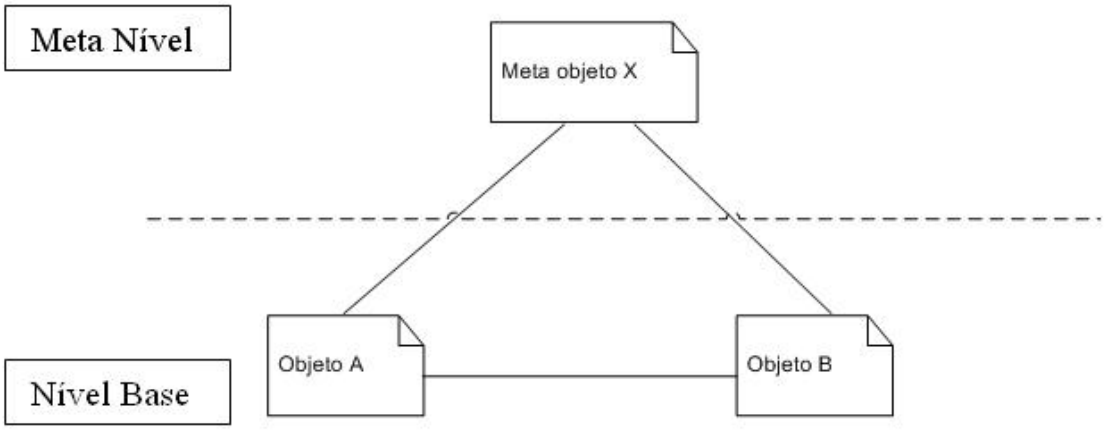


FIGURA 1. Níveis da arquitetura reflexiva.

Fonte: SOUZA, 2001, p.292

2.1.1 Protocolo de Meta-objetos

O uso da reflexão necessita de um mecanismo que permita a troca de mensagens entre o nível base e o meta-nível, e, desta forma, possibilite a tomada de controle do programa por este (LISBOA,1997, p.37). Este mecanismo é denominado protocolo de meta-objetos.

Em um ambiente orientado a objetos, os metadados são organizados em objetos, denominados meta-objetos (FORMAM, 2004, p. 31). O autor ainda define meta-objetos como sendo "um conjunto de objetos presentes no meta-nível, que podem ser usados para representar o programa, bem como sua situação no domínio da aplicação". As entidades do nível base contém propriedades e comportamentos, que se tornarão dados e valores nas entidades do meta-nível.

Podemos definir o protocolo de meta-objetos como uma interface, um meio de comunicação que permite a interação entre os objetos presentes nos dois níveis hierárquicos. Suas responsabilidades são (BARTH, 2000, p.8):

- Definir quais entidades do nível base serão monitoradas pelo meta-nível.

- Determinar as regras presentes na interface de comunicação entre os dois níveis.

- Definir em qual momento o meta-nível assume o controle do sistema.

O protocolo de meta-objetos pode ser dividido em três níveis: Protocolo de introspecção, que possibilita obter informações referentes às classes e objetos do nível base; Protocolo de invocação, responsável pela invocação de métodos de classes do nível base; Protocolo de intercessão, que permite alterar o comportamento dos objetos no nível base (KICZALES,1991, p.5).

De acordo com Lisboa (LISBOA,1997, p.7), o protocolo de meta-objetos possui quatro aspectos básicos:

- Vínculo com o nível base: este aspecto está relacionado à ligação entre os elementos presentes no nível base e no meta nível. Um ponto importante é a cardinalidade desse vínculo, que pode ser estabelecido de forma que um meta-objeto controle um objeto de nível base, ou que mais de um meta-objeto controle um objeto do nível base.

- Reificação: este aspecto refere-se a informações do nível base presentes no meta nível. Esta materialização está relacionada, por exemplo, ao estado das entidades de nível base e mensagens trocadas entre elas.

- Execução: este aspecto está relacionado as operações executadas pelo meta nível.

- Modificação: aspecto relacionado às modificações realizadas pelo meta-nível sobre as entidades do nível base.

Estes aspectos mostram as possíveis atividades do meta nível em relação aos elementos do nível base: associação, identificação da estrutura, interceptação de mensagens, e a possibilidade de alteração da estrutura e manipulação do comportamento dos objetos presentes no nível base.

2.1.2 Interceptação de Mensagens

Os meta-objetos conseguem interceptar a troca de mensagens entre os objetos no nível base, e a partir da análise destas mensagens, podem processar as mesmas, modificando ou não seu conteúdo. A interceptação das mensagens pode ocorrer, por exemplo, quando um método de um determinado objeto é invocado. Desse modo é possível alterar o comportamento do sistema em tempo de execução. Nesse contexto, os meta-objetos assumem o controle sobre as mensagens enviadas pelas entidades do nível base.

A Figura 2, adaptada de (BARTH, 2000, p.14) ilustra a interceptação de uma mensagem emitida por um objeto no nível base, por um objeto no meta-nível. O objeto A invoca um método do objeto B. O meta-objeto intercepta esta mensagem e realiza as computações em cima das informações obtidas, ou seja, sobre os parâmetros passados na chamada do método, alterando ou não os dados da mensagem interceptada. Em seguida, o meta-objeto envia a mensagem ao objeto B no nível base, que executa o método e retorna os dados processados ao objeto A.

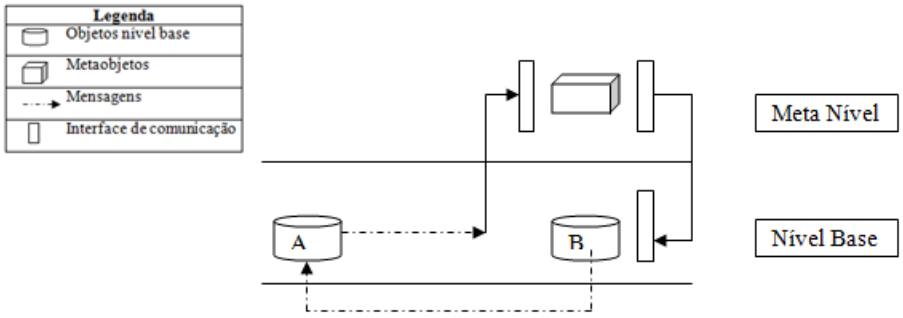


FIGURA 2: Interceptação de mensagens

Fonte: BARTH, 2000, p.14

2.2 Tipos de Reflexão

A reflexão computacional permite que um programa examine a sua estrutura e realize alterações na mesma, possibilitando assim, alterar o modo como se comporta. O meta-nível é o responsável por obter as informações estruturais do sistema (reificação),

e a partir das informações obtidas, poderá manipular os objetos presentes no nível base do sistema, via meta-objetos.

Com base nas informações expostas anteriormente, podemos concluir que a reflexão computacional pode ocorrer de duas formas: reflexão estrutural ou introspecção, e reflexão comportamental, também conhecida como intercessão.

2.2.1 Introspecção

De acordo com Tatsubori, (TATSUBORI,1999, p.15), “a introspecção refere-se ao processo de obter informação estrutural de um programa e usá-la no próprio programa”. Via introspecção é possível obter informações, por meio de sua representação em um meta-nível. Podem ser analisados os elementos de uma classe, tais como atributos, métodos, construtores, modificadores de acesso, interfaces implementadas, classes herdadas, entre outros aspectos.

Outra característica da introspecção é a capacidade de alterar a estrutura de uma classe. É possível, por exemplo, alterar os métodos de uma classe, criar novas estruturas e alterar os modificadores de acesso. Essa prática deve ser realizada com cuidado, pois viola questões relacionadas ao encapsulamento dos objetos existentes.

2.2.2 Intercessão

A reflexão comportamental está relacionada ao processo do próprio programa alterar seu comportamento. Esta alteração é feita por meio da interceptação de ações realizadas no nível base, que podem ser manipuladas e alteradas no meta-nível (SOUZA, 1999, p.7). A interceptação citada pode ocorrer, por exemplo, quando um método é invocado no nível base. O fluxo da mensagem enviada pelo objeto que invocou o método é então desviado para o meta-objeto, que será responsável pelo processo de manipulação da mensagem.

Deste modo, a reflexão comportamental utiliza os meta-objetos no meta-nível para controlar e modificar o comportamento dos objetos aos quais estão relacionados no

nível base. Vale ressaltar que este tipo de reflexão não viola o encapsulamento do objeto, como pode ocorrer na reflexão estrutural.

2.3 CONCLUSÃO

A reflexão computacional é um poderoso recurso fornecido por determinadas linguagens de programação, como, por exemplo, Java, C# e PHP. Por meio do meta-nível, é possível obter informações, alterar a estrutura e manipular o comportamento do sistema em tempo de execução.

Porém, deve-se levar em consideração a perda de performance ao utilizar esta técnica, pois, como foi exposto anteriormente, acessar diretamente os elementos presentes em uma classe é mais rápido que via reflexão.

Na Seção 3, serão discutidos os conceitos do arcabouço *.Net* e a biblioteca de reflexão *System.Reflection* da linguagem C#.

3 ARCABOUÇO .NET E A BIBLIOTECA SYSTEM.REFLECTION

O arcabouço .NET é o ambiente de desenvolvimento e execução de aplicações que utilizam tecnologia Microsoft. Os dois principais elementos presentes no arcabouço são: *Common Language Runtime* (CLR), que é o ambiente de execução do arcabouço, e *a Base Class Library* (BCL), um conjunto de biblioteca de classes que dão suporte ao desenvolvimento de software. Juntos, eles fornecem um ambiente que possibilita a criação, execução, testes e implantação de sistemas em ambiente Windows.

O *System.Reflection* é uma das bibliotecas presentes na BCL. Ela fornece um conjunto de classes que possibilitam, entre outros recursos, examinar a estrutura interna de classes em tempo de execução. Um exemplo de sua utilização pode ser encontrado no próprio Visual Studio (Ferramenta de desenvolvimento de software da Microsoft), em sua função de função auto-completar, conhecida como *IntelliSense*, onde é possível visualizar propriedades e métodos de classes durante o desenvolvimento.

3.1 Arcabouço .NET

Criado em 2002, o arcabouço .NET conseguiu integrar vários ambientes de desenvolvimento, que antes eram executados e gerenciados de forma distinta, entre eles podemos citar o C++,Visual Basic e o ASP3**.**

Os principais recursos disponibilizados pelo arcabouço.NET são (CANNOLY,2007,p.9):

- Interoperabilidade entre linguagens: é possível desenvolver softwares utilizando qualquer linguagem compatível com o arcabouço (C#, C++, *Visual Basic, J#, JScript*).

- Ambiente de execução comum compartilhado entre todas as linguagens: para possibilitar a interoperabilidade entre linguagens, o arcabouço disponibiliza um ambiente de execução comum, onde toda aplicação .NET é executada.

- Biblioteca base de classes disponível por todas as linguagens: o arcabouço disponibiliza um conjunto de módulos e estruturas para o desenvolvimento de software nas linguagens compatíveis com o mesmo.

- Implantação simplificada: não é mais necessário registrar os componentes criados, além de apresentar poucos problemas de implantação em comparação com outras aplicações Windows.

- Segurança: garante segurança com relação ao acesso a código compilado e ao ambiente de execução do arcabouço.

- Desempenho: o código gerado é compilado em uma linguagem intermediária, que mais tarde será convertida em código de máquina. Esta conversão é feita por meio de compiladores *Just-In-Time* (JIT).

A Figura 3, encontrada em (CEMBRANELLI,2003, p.12), apresenta os elementos contidos no arcabouço .NET, que serão descritos nas sub seções seguintes:

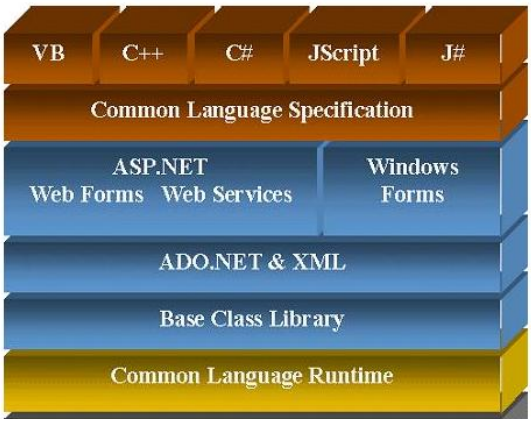


FIGURA 3: Elementos do arcabouço .NET.
Fonte: CEMBRANELLI, 2003, p.12

3.1.1 Common Language Runtime (CLR)

Conforme descrito anteriormente, o CLR é o ambiente de execução do arcabouço .NET. A CLR pode ser considerada uma máquina virtual, pois funciona

como uma interface de comunicação entre o sistema operacional e as aplicações desenvolvidas em linguagens compatíveis com o arcabouço. Podemos compará-la ao

papel que a JVM (*Java Virtual Machine*) desempenha no ambiente Java. Uma das diferenças é que a JVM suporta, originalmente, apenas a linguagem Java (pelo fato de da JVM executar *bytecodes*, ela poderia, em princípio, suportar linguagens diferentes do Java). Já o CLR oferece suporte a mais de uma linguagem, devido ao fato de que os códigos gerados pelas linguagens são convertidos para um mesmo formato (CANNOLY,2007,p.12).

A CLR fornece uma gama de serviços que irão garantir que as aplicações executem de forma adequada e com segurança nos sistema operacionais onde o arcabouço .NET esteja instalado, caracterizando um ambiente de execução gerenciado. Dentre as tarefas realizadas pela CLR podemos citar: gerenciamento de memória, coleta de lixo, *multithreading*, compilação de código, tratamento de erro e suporte a implantação (CEMBRANELLI,2003, p.11).

3.1.2 Commom Language Specification (CLS)

O CLS determina alguns requisitos necessários a uma linguagem de programação para que ela produza código que possa ser executado pela CLR. De acordo com Cannoly (CANNOLY,2007,p.12), as regras da CLS "definem um subconjunto de tipos de dados e construções programáveis que todas as linguagens .NET devem suportar". Esta é uma das características que possibilitam a interoperabilidade de linguagens do arcabouço .NET.

3.1.3 Microsoft Intermediate Language (MSIL)

No ambiente .NET, todo código é compilado para uma linguagem binária intermediária denominada *Microsoft Intermediate Language* (MSIL), sendo considerada uma linguagem de "baixo nível" da plataforma .NET. Sendo assim, todo arquivo executável ou biblioteca (DLL) gerado pelo compilador possui um código executável, e este código está, no .NET, escrito na linguagem MSIL (JUNIOR, 2011, p.16).

É de responsabilidade do CLR converter o código intermediário MSIL em código binário de máquina.

A Figura 4, extraída de (CANNOLY,2007,p.11), ilustra o processo de compilação do arcabouço .NET:

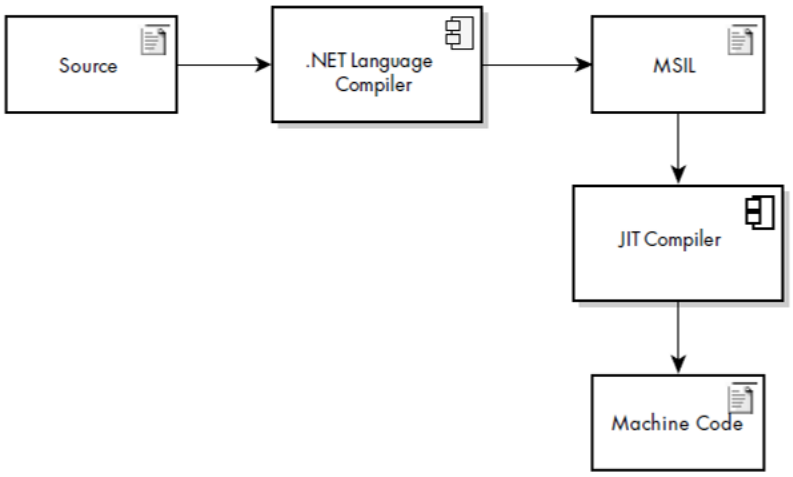


FIGURA 4: Processo de compilação do arcabouço .NET
Fonte: CANNOLY, 2007, p.11

Conforme pode ser observado nessa figura, o código fonte é compilado, e, independente da linguagem utilizada (C++, C#, J#, *Visual Basic* , *JScript*), irá criar o código intermediário MSIL. Este código é então convertido, por meio de um compilador JIT, que se encontra na CLR, e convertido para linguagem binária de máquina.

3.1.4 Base Class Library (BCL)

A *Base Class Library* (BCL), ou biblioteca base de classes, é um conjunto de classes utilizadas para o desenvolvimento de aplicações .NET. Cannoly (CANNOLY,2007,p.13) descreve que esta biblioteca "inclui classes para trabalhar com os tipos básicos e exceções, realizando acesso a dados, e construir interfaces Windows e WEB ".

Todas as linguagens em conformidade com o arcabouço .NET utilizam as mesmas classes presentes na BCL. Por exemplo, a mesma classe e método usados para realizar uma consulta em uma determinada base de dados, utilizando a linguagem VB, será a mesma se utilizarmos a linguagem C#.

Dentre as principais classes presentes na BCL, podemos destacar as seguintes: *Array, Console, Environment, Exception, Math, Object, OperatingSystem, Data, Text, String, Boolean, Byte, Char, Decimal, Double, DateTime, Type e Reflection.*

3.2 Reflexão Computacional em C#

A linguagem de programação C# provê o mecanismo de reflexão computacional, por meio da biblioteca *System.Reflection,* presente no arcabouço .NET. Porém, essa biblioteca oferece suporte apenas à reflexão estrutural, que permite inspecionar os elementos de uma classe ou *assembly,* permitindo obter informações sobre os atributos, métodos, modificadores de acesso de uma classe, possibilitando também instanciar objetos, definir valores de atributos e invocar métodos da mesma.

A reflexão comportamental não é disponibilizada pela biblioteca *System.Reflection*. Não é possível, por exemplo, interceptar a troca de mensagens entre objetos e alterar o comportamento de uma determinada operação em tempo de execução, característica esta presente em outras linguagens de programação. No entanto, existem bibliotecas alternativas capazes de realizar tanto a reflexão estrutural como comportamental em C#. Podemos citar o Projeto RAIL (*Runtime Assembly Instrumentation Library*), que possui um mecanismo capaz de reescrever códigos de assemblies em tempo de execução (CABRAL, 2004, p. 1). Esta questão não será detalhada no presente trabalho, que abordará apenas a reflexão estrutural disponibilizada pelo arcabouço .NET.

Conforme Liberty, (LIBERTY, 2007, p.475), a reflexão em .NET, e na maioria das linguagens OO que oferecem suporte a reflexão, é geralmente utilizada para realizar uma das quatro tarefas a seguir:

- Visualização de metadados: pode ser usado por ferramentas e utilitários que desejam mostrar metadados.

- Descoberta de tipos: permite examinar os tipos presentes em um *assembly*, interagir com eles ou instanciar seus tipos.

- Invocação dinâmica: permite ao programador invocar propriedades e métodos em objetos dinamicamente instanciados, baseado na descoberta do tipo.

- Criação de tipos em tempo de execução (*reflection emit*): permite criar tipos em tempo de execução e, em seguida, usá-los para realizar tarefas.

3.3 Biblioteca System.Reflection

Como mostrado na Seção 3.1, o .NET suporta reflexão estrutural amplamente por meio da biblioteca *System.Reflection*. De acordo com o manual da Microsoft, (MICROSOFT, 2011), "a biblioteca *System.Reflection* contém tipos que permitem recuperar informações de *assemblies*, módulos, membros, parâmetros, e outras entidades do código gerenciado, examinando seus metadados". O mesmo ainda cita a capacidade de invocar métodos e instanciar objetos em tempo de execução, por meio da manipulação de instâncias de tipos.

A Tabela 1 apresenta as principais classes da biblioteca *System.Reflection*, descrevendo as responsabilidades de cada uma delas (MICROSOFT, 2011):

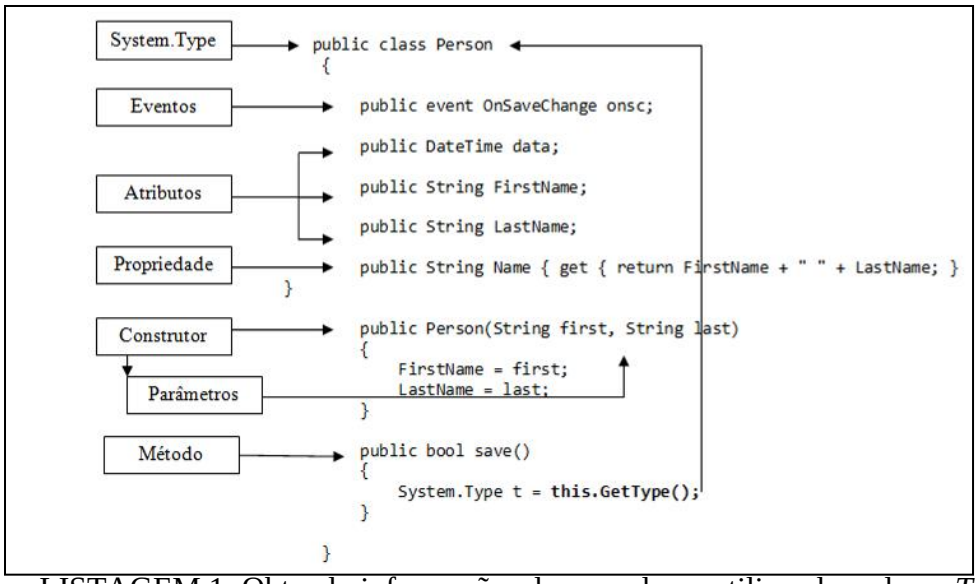
Classe	Responsabilidade
Assembly	Definir e carregar *assemblies*; carregar módulos que estão listados no manifesto do *assembly*; localizar um tipo a partir de um assembly e criar uma instância do mesmo.
Module	Descobrir informações de um módulo.
ConstructorInfo	Descobrir informações como o nome, parâmetros, modificadores

	de acesso e detalhes de implementação de um construtor.
MethodInfo	Descobrir informações como o nome, tipo de retorno, parâmetros modificadores de acesso e detalhes de implementação de um método.
MemberInfo	Obter informações sobre os atributos de um membro e prover acesso aos metadados do membro
FieldInfo	Descobrir informações como o nome, modificadores de acesso e detalhes de implementação de um campo, possibilitando também obter ou definir valores de um campo.
EventInfo	Descobrir informações como nome do manipulador de eventos do tipo de dados, atributos personalizados, tipos declarados, e adicionar ou remover manipuladores de eventos.
PropertyInfo	Descobrir informações como o nome, tipo de dados, tipos declarados, tipo e status de somente leitura ou escrita de uma propriedade, além obter ou definir valores de propriedade.
ParameterInfo	Descobrir informações como o nome de um parâmetro, tipo de dados, se um parâmetro é de entrada ou de saída, bem como a posição do parâmetro em uma assinatura do método.

TABELA 1: Principais classes da biblioteca *System.Reflection*

Uma das classes mais importantes da biblioteca é a classe *Type*. Considerada como o ponto central da biblioteca de reflexão do arcabouço .NET, esta classe permite acessar as informações de uma determinada classe, herdando métodos e atributos da classe *MemberInfo*, sendo uma porta de entrada para a maior parte das manipulações usando reflexão. Segundo Liberty, (LIBERTY, 2007, p.456) "A classe *Type* é o coração das classes de reflexão. Ela encapsula a representação do tipo de um objeto, sendo o primeiro caminho para acesso aos metadados."

A Listagem 1, adaptada de (LIBERTY, 2007, p.470) apresenta os membros de uma classe e a forma como é feita a reflexão da mesma por meio da classe *Type*. Cada instância de uma determinada classe pode invocar o método *GetType*(), que irá retornar o tipo relacionado ao objeto.



LISTAGEM 1: Obtendo informações de uma classe utilizando a classe *Type*

3.3.1 Instanciação Dinâmica de Objetos

Existem duas maneiras de se criar instâncias em tempo de execução. A primeira utiliza a classe *System.Reflection.ConstructorInfo*, que, por meio do método *Invoke*(), recebendo como parâmetro um array de objetos referentes aos parâmetros do construtor, retorna uma instância da classe em questão. A Listagem 2 apresenta um trecho de código onde uma instância da classe "*Class1*" é criada utilizando a classe *System.Reflection.ConstructorInfo*:

```
System.Type t = System.Type.GetType("Class1");

System.Reflection.ConstructorInfo constructorInfo =
      t.GetConstructor(new Type[] { });

Object obj = constructorInfo.Invoke(new object[]{});
```

LISTAGEM 2: Criando instância de uma classe utilizando *System.reflection.ConstructorInfo*

A outra maneira de criar instâncias em tempo de execução é utilizando o método *CreateInstance*(), da classe *System.Activator*. Este método recebe como parâmetro um objeto da classe *System.Type*, referente a classe a qual se deseja criar uma instância, e retorna um objeto da classe. A Listagem 3, extraída de Troelsen, (TROELSEN, 2008, p.

540), apresenta um trecho de código onde uma instância da classe "*Class1*" é criada utilizando a classe *System.Activator*:

```
System.Type t = System.Type.GetType("Class1");

Object obj = System.Activator.CreateInstance(t);
```

LISTAGEM 3: Criando instância de uma classe utilizando *System.Activator*

3.3.2 Invocação Dinâmica de Métodos

A biblioteca *System.Reflection* permite invocar métodos em tempo de execução. Esta invocação pode ser feita por meio de uma instância da classe criada dinamicamente, ou obtendo informações da mesma via a classe *Type*. Confome Marshall, (MARSHALL,2005, p.373), na reflexão, há duas abordagens para chamar um método dinamicamente: *MethodInfo.Invoke* e *Type.InvokeMember*, onde *MethodInfo.Invoke* é a solução mais simples.

A Listagem 4 apresenta um trecho de código com a invocação dinâmica de métodos utilizando as duas abordagens decritas acima.

```
class Program
        {
            static void Main(string[] args)
            {
                //Cria uma instancia da classe a ser refletida.
                Class1 objClass1=new Class1();

                //Invoca o método EscreverTexto  da classe "Class1"
                //utilizando a classe System.Reflection.MethodInfo

                System.Reflection.MethodInfo methodInfo =
objClass1.GetType().GetMethod("EscreverTexto");
                methodInfo.Invoke(objClass1,null);


                //Invoca o método EscreverTexto  da classe "Class1"
                //utilizando a classe System.Reflection.MethodInfo

                Type t = objClass1.GetType();
                t.InvokeMember("EscreverTexto",
BindingFlags.InvokeMethod,null,objClass1,null);

            }
        }

        class Class1
        {
            public void ExcreverTexto()
            {
                Console.WriteLine("Método invocado com sucesso.");
            }
        }
```

LISTAGEM 4: Código referente a invocação dinâmica de métodos

Como pode ser visto, inicialmente foi realizada a invocação dinâmica utilizando a classe *MethodInfo*, que recebe o nome do método a ser invocado via a classe *Type* e invoca o mesmo passando como parâmetro o objeto da classe instanciado anteriormente. Como o método não precisa de nenhum parâmetro, o segundo argumento do método *Invoke* é passado como *null*.

Na sequência do código, o mesmo método foi invocado, agora utilizando o método *InvokeMember*, da classe *Type*. Este método necessita de uma sinalização, informando o membro da classe que será invocado, no caso, um método.

3.3.3 Criação Dinâmica de Tipos

A biblioteca de reflexão do arcabouço .NET possibilita a criação de *assemblies* em tempo de execução, por meio da biblioteca *System.Reflection.Emit*. Estes *assemblies* são gerados em memória, por meio de tipos presentes na biblioteca *System.Reflection.Emit*, que possibilita, em um segundo momento, salvar os mesmos em um disco e utilizá-los para realizar determinadas tarefas do sistema (TROELSEN, 2008, p. 648).

A utilização da biblioteca *System.Reflection.Emit* abstrai o processo de criação de tipos dinâmicos, possuindo estruturas para a construção de classes, atributos, métodos, eventos, entre outros elementos presentes em um *assembly*. Porém, a construção de classes complexas exige um conhecimento da linguagem intermediária MSIL, pois o código gerado utiliza instruções da mesma.

A Tabela 2 apresenta os principais membros presentes na biblioteca *System.Reflection.Emit* (TROELSEN, 2008, p. 649):

Membro	**Função**
AssmblyBuilder	Criação de *assembly* (DLL ou exe) em tempo de execução.
ModuleBuilder	Definição de um conjunto de módulos de um *assembly*
TypeBuilde	Criação de classes, interfaces e estruturas de um módulo.
MethodBuilder	Criação métodos em tempo de execução.
PropertyBuilder	Criação de propriedades em tempo de execução.
ConstructorBuilder	Criação de construtores em tempo de execução.

ILGenerator	Emissão de OpCodes MSIL em um determinado tipo.
OpCodes	Fornece atributos que são mapeados em instruções MSIL

TABELA 2: Membros da biblioteca *System.Reflection.Emit*

Dentre as utilidades da criação de tipos em tempo de execução, podemos destacar:

- Alterar a estrutura interna de *assemblies* já criados, utilizando-o em memória para a realização de outras tarefas.
- Criar ferramentas e controles baseados em inputs do usuário (TROELSEN, 2008, p. 648).
- Gerar classes de entidades baseadas em tabelas de um banco de dados.

3.4 Conclusão

Nesta seção, foram apresentados os principais elementos presentes no arcabouço .NET, responsável por gerenciar o ambiente de execução e construção das aplicações desenvolvidas na tecnologia Microsoft.

A seção também destacou a biblioteca de reflexão do arcabouço .NET, *System.Reflection*. A reflexão presente no arcabouço .NET permite visualizar a estrutura interna de um determinado módulo em tempo de execução. Além de examinar o conteúdo destas estruturas, a biblioteca *System.Reflection* possui classes e métodos capazes de instanciar objetos, invocar métodos e até criar tipos durante a execução de determinada aplicação. Este recurso é utilizado, por exemplo, no arcabouço de persistência NHibernate (HIBERNATE, 2011), que usa biblioteca *System.Reflection.emit* para criar *proxies* de objetos em tempo de execução, seguindo o padrão de projeto Dynamic Proxy.

Na próxima seção, será apresentado o conceito de arquitetura em camadas, um recurso muito utilizado atualmente no desenvolvimento de softwares, que possibilita o desacoplamento das classes do sistema e alta coesão das mesmas.

4 ARQUITETURA DE SOFTWARE EM CAMADAS

O processo de desenvolvimento de software vem evoluindo com o passar do tempo, na medida em que novas tecnologias e paradigmas de linguagens de programação surgem. No período que compreende as décadas de 70 e meados de 80, o paradigma estruturado predominava nas equipes de desenvolvimento de software (DEITEL, 2002, p. 47). Hoje, o paradigma orientado por objetos é o mais utilizado, possuindo recursos como abstração, encapsulamento, polimorfismo, que possibilitam uma maior flexibilidade e reusabilidade de código no desenvolvimento das aplicações, sejam elas de pequeno ou grande porte.

No entanto, para se desenvolver um software de qualidade, não basta apenas escrever classes e métodos sem pensar em como estes elementos estarão estruturados no projeto. Hoje nas empresas, há uma grande necessidade de se reaproveitar determinados procedimentos em sistemas distintos, com objetivo de padronizar e ganhar tempo no processo de desenvolvimento. Além disto, é importante lembrar que novos requisitos podem ser incorporados ao sistema, e estas novas funcionalidades não devem afetar o que já foi construído anteriormente. Esta idéia segue o princípio conhecido como Abertura-Fechamento (OCP – *Open Closed Principle*): *"Entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação."* (MEYER, 1997, p.23)

Assim, é desejável que os desenvolvedores de software construam sistemas que possam ser reutilizados, e,que ao mesmo tempo facilitem o seu processo de manutenção. O recurso mais importante para obter essas facilidades é o módulo. Na literatura há várias definições de módulo. A seguir algumas delas:

- Um módulo é um arquivo que contém uma parte executável do seu programa (AITKEN, 1991, p.34).
- Menor unidade na qual o sistema pode ser decomposto (MEYER, 1997, p.153).

Sendo assim, um módulo dentro de um sistema de software pode ser uma classe, um método, um pacote contendo determinadas classes.

Esta arquitetura em módulos deve estar alinhada com dois conceitos muito importantes no que diz respeito ao desenvolvimento dos módulos de um sistema: alta coesão e baixo acoplamento.

Coesão de um módulo diz respeito ao relacionamento entre os membros de um módulo, sejam eles classes ou métodos. Um módulo com alta coesão deve ser responsável por realizar uma única tarefa, e seus membros devem estar estruturados de forma a alcançar os objetivos dessa tarefa. Segundo Allen, (ALLEN,1999, p.6), coesão "é uma medida de quão fortemente relacionadas estão as funcionalidades do código de um módulo de software".

De acordo com Ormandjieva, (ORMANDJIEVA,2005,p.3), acoplamento de módulos "é uma medida da interdependência entre os componentes de um sistema de software que descreve a natureza e a extensão das conexões entre os elementos do sistema". Quanto menos uma classe conhecer ou depender de outra para realizar suas tarefas, menos acoplada ela estará. Essa característica possui grande influência na manutenção das mesmas, pois para realizar tal tarefa em uma determinada classe com alto acoplamento, será necessário conhecer as classes com que ela se relaciona. Além disto, uma alteração em uma destas classes poderá causar impacto nas demais. Podemos observar que um conceito está relacionado ao outro. Classes altamente coesas tendem a possuir baixo acoplamento, pois um determinado módulo não precisa se relacionar com muitos outros, ou até nenhum, para realizar suas funções.

4.1 Arquitetura em Camadas

A arquitetura em camadas é uma forma de separar os módulos de uma aplicação de acordo com o tipo de funcionalidade ou serviço que cada um deles presta ao sistema. Este modelo, muito utilizado em sistemas orientados a objetos, visa aumentar a modularidade, manutenibilidade, extensibilidade e a portabilidade do software (SANTOS, 2006, p.47).

Os primeiros projetos desenvolvidos neste contexto eram compostos de duas camadas: a camada de interface e a camada de negócio. Na camada de interface estão presentes as classes de interface gráfica. É onde o usuário interage com o sistema, realizando requisições e salvando informações. A camada de negócio é responsável pelas regras de negócio do sistema. A Figura 5 mostra esta arquitetura. Note que o fluxo de mensagens parte sempre da camada de interface para a camada de negócios.

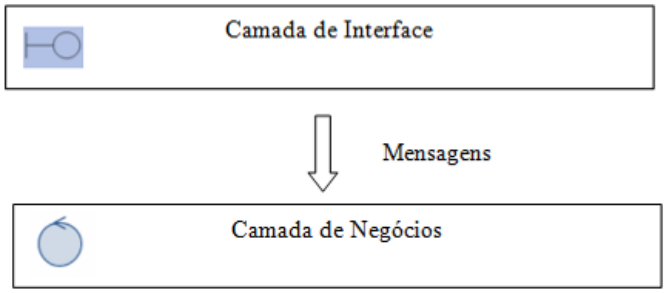


FIGURA 5: Arquitetura em duas camadas

Fonte: Próprio autor.

No entanto, esta arquitetura apresenta alguns problemas. Um deles está relacionado à persistência dos dados. O processo de armazenamento de dados nesta arquitetura fica embutido na camada de Negócios, o que não é uma boa prática e acaba prejudicando a modularidade do sistema. A arquitetura em três camadas visa solucionar esta deficiência, que será apresentado na Seção 4.2.

4.2 Modelo em Três Camadas

A arquitetura em três camadas manteve as camada de interface e negócios, conhecida como BLL (*Business Logic Layer*), presentes na arquitetura de duas camadas. Essa arquitetura propõe a existência de uma camada para persistência de dados, conhecida como DAL (*Data Access Layer*) (SANTOS, 2006, p.49).

A camada de persistência tem por objetivo persistir os dados da aplicação em algum tipo de repositório, seja ele um arquivo em um diretório local, mídia digital ou banco de dados, este último sendo o mais utilizado para este fim. Além disso, possui como responsabilidade centralizar o acesso aos dados do sistema. As operações CRUD (*Create, Recover*, *Update, Delete*) são desenvolvidas dentro desta camada.

O fluxo de mensagens entre as três camadas deve ocorrer sempre em uma única direção, em especial, sempre da camada de interface para a camada de negócios, e, da camada de negócios para a camada de persistência. Deste modo, a camada de negócios

funciona como uma ponte entre as camadas de interface e de persistência, contendo as regras de negócios e realizando validações sobre os dados.

Seguindo este raciocínio, as camadas de persistência e de negócios devem ser construídas de forma independente da camada de interface. Essa característica irá permitir que a mesma possa ser desenvolvida em vários tipos de interfaces distintas, sem impactar nas demais.

A Figura 6, extraída de (PAULI, 2007, p.32) , apresenta uma arquitetura em três camadas: Interface, Negócios e Persistência. Note que a camada de interface possui três alternativas: WEB, *Mobile* e *Desktop*.

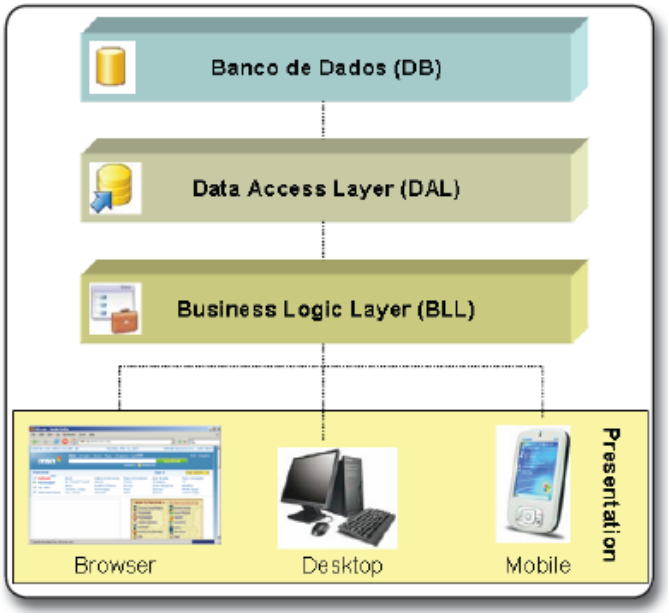


FIGURA 6: Arquitetura em três camadas

Fonte: PAULI, 2007, p.32

Apesar das melhorias referentes a modularidade dos sistemas em comparação com o modelo em duas camadas, esta arquitetura ainda apresenta algumas limitações. Não há, por exemplo, a separação de classes responsáveis pelo acesso a recursos do sistema operacional, *hardware* ou a outros sistemas. Sendo assim, faz-se necessário a existência de uma camada adicional, conhecida como camada de Sistema, responsável pelo empacotamento das classes de sistema que fornecem acesso a recursos externos a aplicação (FERREIRA, 2006, p.50). Essa camada será discutida na Seção 4.3.

4.2.1 Vantagens e Desvantagens da Arquitetura de 3 Camadas

A arquitetura em três camadas apresenta as seguintes vantagens (PAULI, 2007, p.33):

- interface Independente: A camada de interface contém apenas código referente à própria interface, não possuindo conhecimento a respeito de regras de negócio nem informações referentes à persistência, como tabelas, ou tipo do banco de dados. Sua função é apresentar os dados ao usuário e manipulá-los. Devido a esta característica, podemos dizer que a aplicação cliente é *thin-client* (cliente leves), proporcionando um elevado nível de abstração;

- independência de tipo de aplicação cliente: como o acesso a dados é centralizado, podemos compartilhar esse código de acesso entre vários tipos de aplicações cliente, como *Web, Desktop e Mobile*;

- migração facilitada: O fato de a arquitetura estar dividida em camadas facilita a migração do sistema. Se quisermos , por exemplo, migrar utilizar um outro tipo de banco de dados para persistir os dados, apenas uma camada (a DAL) precisaria sofrer alguns ajustes, as outras camadas não enxergam esse acesso;

- reaproveitamento de código: o código da camada de negócios (BLL) por exemplo, não é replicado caso tenhamos vários tipos de aplicativos cliente ou caso tenhamos múltiplas interfaces fazendo uso dos mesmos objetos;

- manutenção facilitada: como os aplicativos estão em camadas, se houver um erro ou for necessário fazer uma manutenção / correção, normalmente apenas uma camada é afetada, o que também facilita o processo de implantação;

Este modelo também apresenta algumas desvantagens, por exemplo, o uso das camadas em sistemas de pequeno porte. Em alguns casos, a criação das pode complicar um sistema relativamente simples.

Outra desvantagem deste modelo é não possuir uma camada para acesso a dados externos do sistema. Santos, (SANTOS, 2006, p.47), destaca que esta arquitetura "não apresenta o empacotamento de um conjunto de funcionalidades que atendam, por exemplo, as chamadas ao sistema operacional, acesso a rede, etc."

4.3 Arquitetura em N Camadas

A arquitetura em três camadas foi evoluindo ao longo do tempo. Na medida em que os sistemas foram crescendo e se tornando mais complexos, sentiu-se a necessidade de adicionar novas camadas ao projeto, a fim de garantir maior qualidade e modularidade dos mesmos. Atualmente, arquiteturas utilizando N camadas são adotadas na construção de determinadas aplicações, de acordo com a sua necessidade.

A Figura 7, extraída de Ambler (AMBLER, 1998, p.97), apresenta uma arquitetura em cinco camadas. Neste modelo, além das já discutidas anteriormente (interface, negócios e persistência), ele também possui uma camada com classes de sistema, cuja responsabilidade é prover acesso a recursos externos, tais como sistema operacional ou *hardware*. A outra camada desse modelo é a de usuários, referente aos usuários que utilizam o sistema, que, segundo Ambler (AMBLER, 1998, p.98), é uma peça fundamental no sistema como um todo.

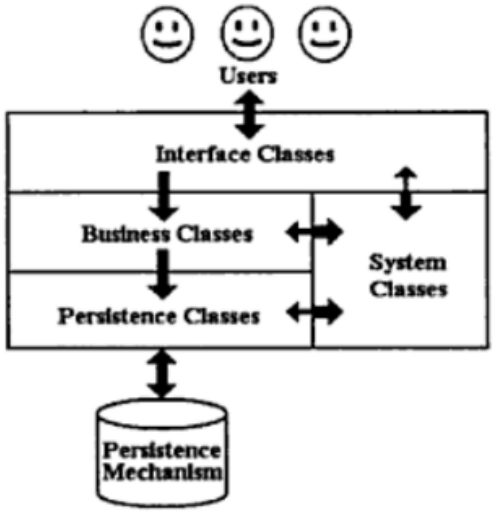


FIGURA 7: Arquitetura em 5 camadas

Fonte: AMBLER, 1998, p.98

O planejamento relacionado a quantas camadas e como elas serão estruturadas irá depender das funcionalidades do sistema, não havendo uma regra específica para cada uma (SOUZA, 2006, p.48).

Geralmente, quanto mais complexo e robusto for um sistema, mais camadas deverão ser construídas a fim de garantir um alto grau de coesão e acoplamento. Como destacado anteriormente, essas características facilitarão na manutenção e extensão do sistema.

4.4 Conclusão

Nesta seção, foram apresentados os conceitos da arquitetura em camadas no desenvolvimento de software. O modelo em três camadas é bastante adotado em equipes de desenvolvimento de softwares, pois permite desacoplar ainda mais os módulos do sistema, em comparação com a arquitetura de duas camadas. No entanto, nada impede que a aplicação possua outras camadas, e isso será definido de acordo com as funcionalidades do mesmo.

A Seção 5 irá tratar da parte prática deste trabalho. Será desenvolvida uma aplicação Web, com o objetivo de demonstrar a vinculação de dados entre as camadas

de interface e negócios, em páginas construídas dinamicamente. Para a construção da aplicação, serão utilizados os conceitos apresentados nas seções anteriores.

5 DESENVOLVIMENTO DA APLICAÇÃO PROPOSTA

A gestão eletrônica de documentos (GED) é uma atividade que cresce a cada dia em ambientes corporativos. O armazenamento de documentos em mídias digitais permite que montantes de papéis armazenados sejam descartados, proporcionando ganhos relacionados a espaço físico e controle dos mesmos. Além destes benefícios, o uso do GED como meio de armazenamento está em concordância com o modelo de TI Verde, que propõe uma prática sustentável nas organizações.

A aplicação proposta se baseia no conceito do GED. No processo, documentos são enviados pelas empresas e armazenados em uma base de dados. Após este processo, uma equipe de auditores é responsável por analisar os documentos enviados eletronicamente, cuja atividade é extrair os dados dos documentos em um formulário específico. No fim da análise, os dados são processados e salvos no banco de dados, podendo servir de base para a geração de relatórios gerenciais.

O fato de cada documento possuir campos específicos, por exemplo, documentos trabalhistas possuem informações referentes aos funcionários, exige que a página de análise dos mesmos seja construída dinamicamente, de acordo com os campos presentes nos documentos e configurados previamente no sistema. Deste modo, o formulário onde o usuário irá entrar com os dados extraídos será específico para cada documento.

Nesse contexto, a reflexão computacional tem por objetivo transferir os dados presentes no formulário dinâmico entre as camadas de interface e negócios (BLL) do sistema. Como cada campo do formulário é criado em tempo de execução, o uso da reflexão torna-se uma forma eficiente de vincular os dados entre camadas da aplicação, eliminando a necessidade de se criar estruturas condicionais para instanciar objetos de acordo com o documento escolhido.

A idéia básica do uso da reflexão nesta aplicação proposta é criar um meio automático de captura e transferência de dados entre as camadas do sistema, cujos passos são, resumidamente:

1. identificar o objeto de negócio correspondente e criar uma instância do mesmo.

2. atribuir os valores do formulário aos atributos do objeto.

3. invocar o método responsável pelo processamento dos dados obtidos, conforme a regra de negócio estabelecida.

Esta seção descreve os objetivos do sistema proposto, suas funcionalidades, arquitetura e os detalhes mais importantes de sua implementação. A notação UML foi utilizada para a descrição do sistema, utilizando o Microsoft Visio 2010 como ferramenta de modelagem dos diagramas. Para armazenamento dos dados, foi utilizado o banco de dados SQL Server 2008.

5.1 ESPECIFICAÇÃO DOS REQUISITOS

A especificação dos requisitos será composta pelo diagrama de caso de uso, suas descrições, atores, e pela descrição das interfaces do usuário. Este modelo foi baseado nos artefatos de especificação de requisitos de software (ERSw) proposto por Paula (PAULA, 2001, p.78).

5.1.1 Casos de Uso

O sistema GED proposto possui dois atores principais:

Usuário básico: funcionário de determinada empresa responsável por enviar os arquivos de imagem dos documentos.

Auditor: responsável por cadastrar novos tipos de documentos e analisar os documentos enviados.

A Figura 8 demonstra os casos de uso e o diagrama de caso de uso do sistema GED proposto:

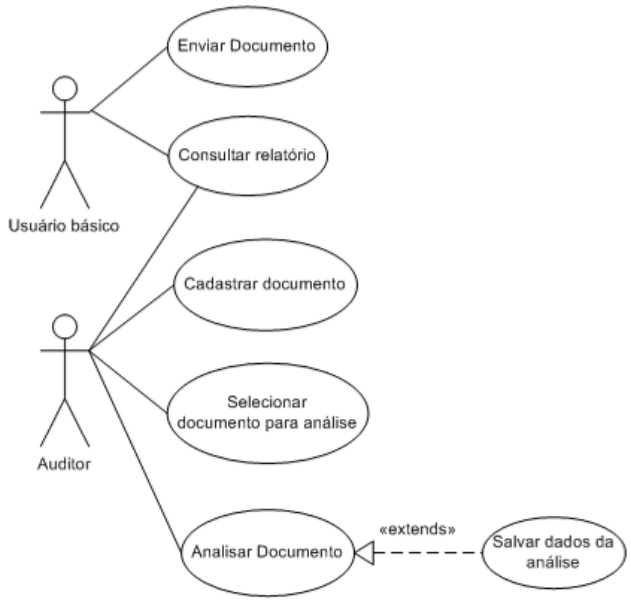


FIGURA 8: Diagrama de Casos de Uso do sistema GED.

Fonte: Próprio autor.

A Tabela 3 descreve os casos de uso do sistema:

Caso de Uso	Descrição	Ator
Enviar Documento	Funcionalidade que permite ao usuário enviar a um determinado documento,via upload de arquivo.	Usuário básico
Cadastrar Documento	Funcionalidade que permitirá ao usuário cadastrar novos tipos de documentos, informando seus respectivos campos de análise.	Auditor
Selecionar Documento para Análise	Funcionalidade que permitirá ao usuário selecionar um determinado documento dentre a lista de documentos aguardando análise.	Auditor
Analisar Documento	Funcionalidade que permitirá ao usuário analisar os documentos enviados, extraindo os dados dos mesmos nos campos de análise.	Auditor
Consultar Relatórios	Funcionalidade que permitirá ao usuário consultar relatórios referentes à análise dos documentos.	Auditor / Usuário básico

Salvar Dados da Análise	Funcionalidade responsável por processar e gravar os dados da análise na base de dados.	Sistema

TABELA 3: Descrição dos casos de uso

5.1.2 Fluxo do processo de negócios

O fluxo do processo de análise de documentos se inicia com o envio de um documento pelo Usuário básico. O Auditor então seleciona o documento e inicia a análise do mesmo, visualizando o documento e extraindo suas informações no formulário gerado. Ao finalizar a análise, os dados são processados e armazenados no banco de dados, servindo de base para a construção de relatórios gerenciais.

A Figura 9 apresenta o diagrama de atividades referente ao processo de análise de documentos do sistema. Ele apresenta as atividades realizadas pelos dois atores do sistema (Usuário básico e Auditor).

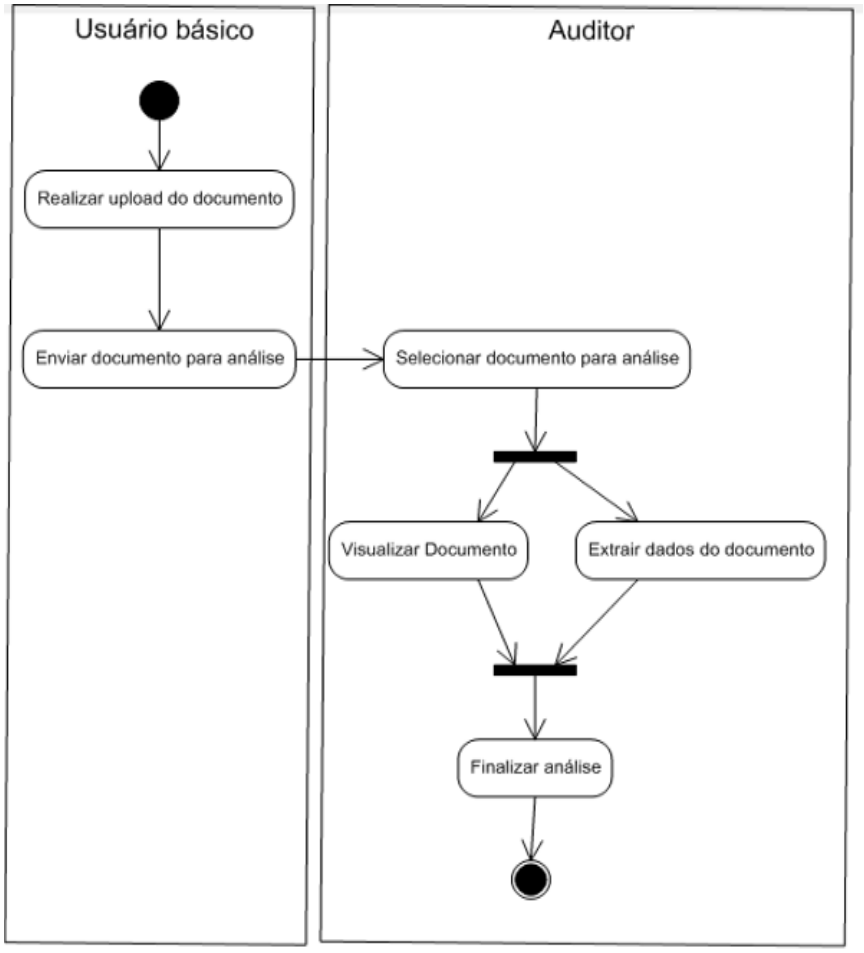


FIGURA 9: Diagrama de atividades do processo de análise de documentos

Fonte: Próprio autor.

5.1.3 Interfaces de Usuário

O sistema GED possui as seguintes interfaces de usuário: Login, Menu Principal, Envio de Documento, Cadastro de Documento, Seleção de Documentos para Análise, Análise de Documento, Consulta de Relatórios. Segue a descrição das interfaces:

- Login: Tela inicial da aplicação, responsável pela autenticação do usuário. A Tabela 4 descreve o comando desta interface e sua respectiva ação. A Figura 10 apresenta sua interface.

Comando	**Ação**
Entrar	Verifica se o usuário é válido e redireciona para a tela do Menu Principal

TABELA 4: Comandos da Interface *Login*

FIGURA 10: Interface *Login*

Fonte: Próprio autor.

- *Menu Principal*: Tela inicial da aplicação, onde estão presentes os menus do sistema. A Tabela 5 descreve os comandos desta interface e suas respectivas ações. A Figura 11 apresenta sua interface.

Comando	**Ação**
Envio de Documento	Exibe a interface de usuário *Envio de Documento.*
Cadastro de Documento	Exibe a interface de usuário *Cadastro de Documento.*
Análise de Documento	Exibe a interface de usuário *Seleção de Documentos para Análise.*
Relatório	Exibe a interface de usuário *Consulta de Relatórios.*

TABELA 5: Comandos da Interface *Menu Principal*

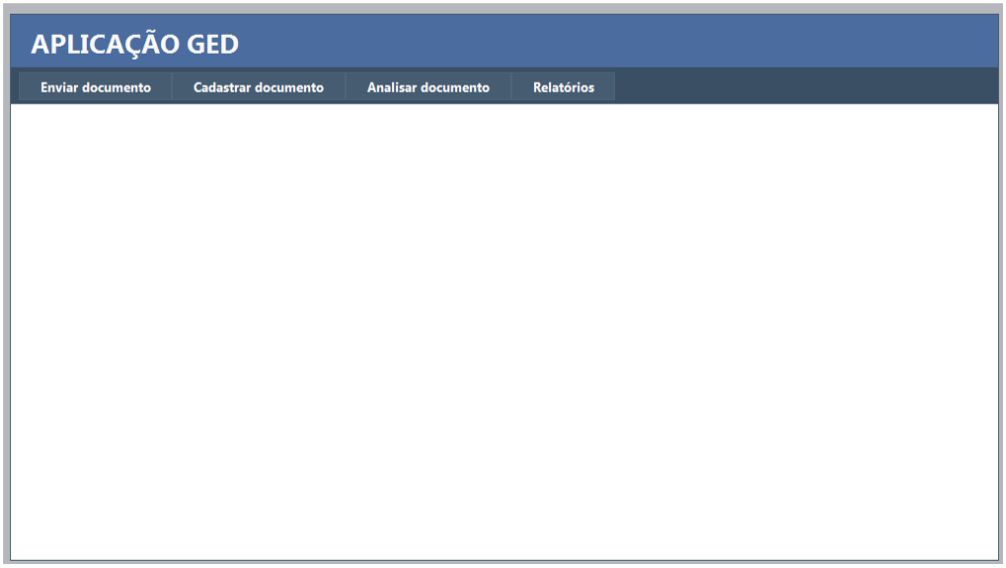


FIGURA 11: Interface *Menu Principal*

Fonte: Próprio autor.

- *Envio de Documento*: Interface onde o usuário seleciona o tipo de documento que será enviado, o mês de competência do documento e o arquivo que será enviado. A Tabela 6 descreve os comandos desta interface e suas respectivas ações. A Figura 12 apresenta sua interface.

Comando	**Ação**
Procurar	Seleciona o arquivo a ser enviado.

Enviar	Envia o documento selecionado.

TABELA 6: Comandos da interface *Envio de Documento*

FIGURA 12: Interface *Envio de Documento*

Fonte: Próprio autor.

- *Cadastro de Documento*: Interface onde o usuário cadastra novos tipos de documentos, informando seus respectivos campos para análise. Esses campos serão utilizados na construção do formulário dinâmico da interface *Análise de Documento*. A Tabela 7 descreve os comandos desta interface e suas respectivas ações. A Figura 13 apresenta sua interface.

Comando	**Ação**
Adicionar Campo	Abre painel onde o usuário informa os dados do novo campo para análise.
Salvar novo Campo	Salva o novo campo em uma tabela temporária.
Gravar Documento	Salva os dados do novo documento na base de dados.

TABELA 7: Comandos da interface *Cadastro de Documento*

FIGURA 13: Interface *Cadastro de Documento*

Fonte: Próprio autor.

- *Seleção de Documentos para Análise*: Interface onde são apresentados os documentos enviados que ainda não foram analisados. A Tabela 8 descreve o comando desta interface e sua respectiva ação. A Figura 14 apresenta sua interface.

Comando	**Ação**
Analisar	Redireciona para a página onde o usuário irá analisar a imagem do documento.

TABELA 8: Comandos da interface *Seleção de Documentos para Análise*

FIGURA 14: Interface *Seleção de Documentos para Análise*

Fonte: Próprio autor.

- *Análise de Documento*: Interface onde o usuário visualiza a imagem do documento e extrai as informações do mesmo nos campos do formulário associados ao documento. A Tabela 9 descreve os comandos desta interface e suas respectivas ações. A Figura 15 apresenta sua interface.

Comando	**Ação**
Analisar	Recupera o documento e o exibe em um painel. Recupera os campos associados ao documento e constrói o formulário dinâmico.
Zoom (click na imagem)	Aumenta o tamanho da imagem do documento
Finalizar Validação	Salva os dados da análise na base de dados.

TABELA 9: Comandos da interface *Análise de Documento*

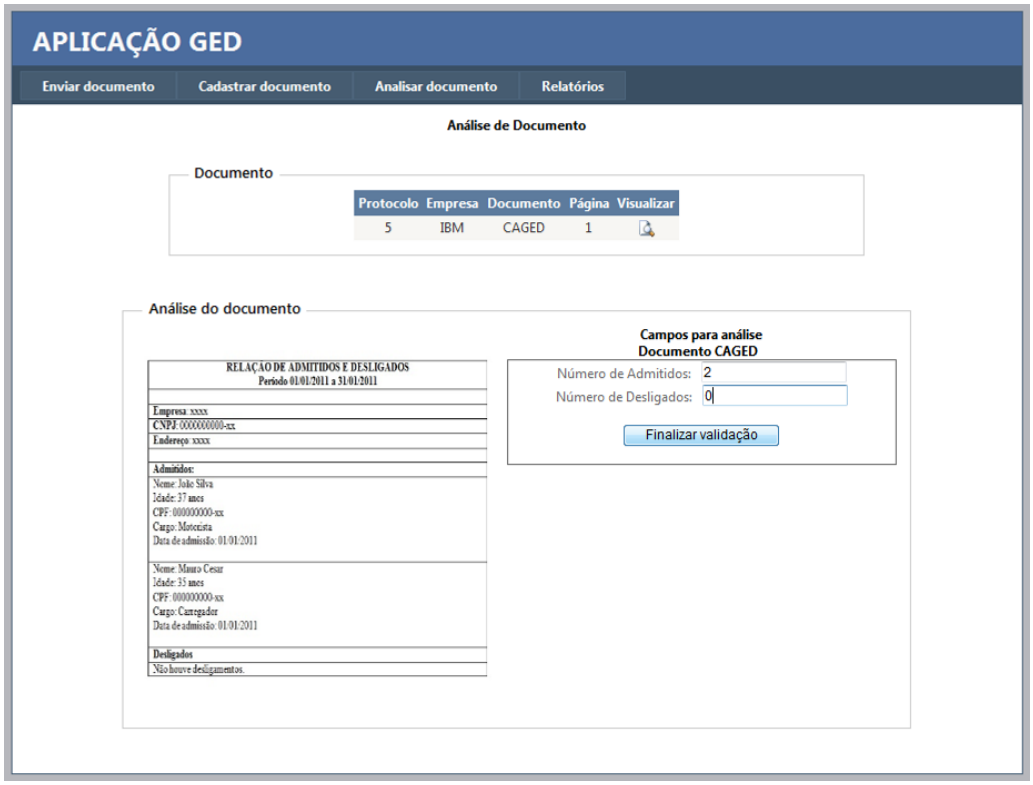


FIGURA 15: Interface *Análise de Documento*

Fonte: Próprio autor.

- *Relatório*: Interface onde o usuário acessa os relatórios resultantes da análise dos documentos. A Tabela 10 descreve o comando desta interface e sua respectiva ação. A Figura 16 apresenta sua interface.

Comando	**Ação**
Consultar	Gera o relatório de acordo com o tipo de documento (podendo selecionar a opção todos os documento).

TABELA 10: Comandos da interface *Relatório*

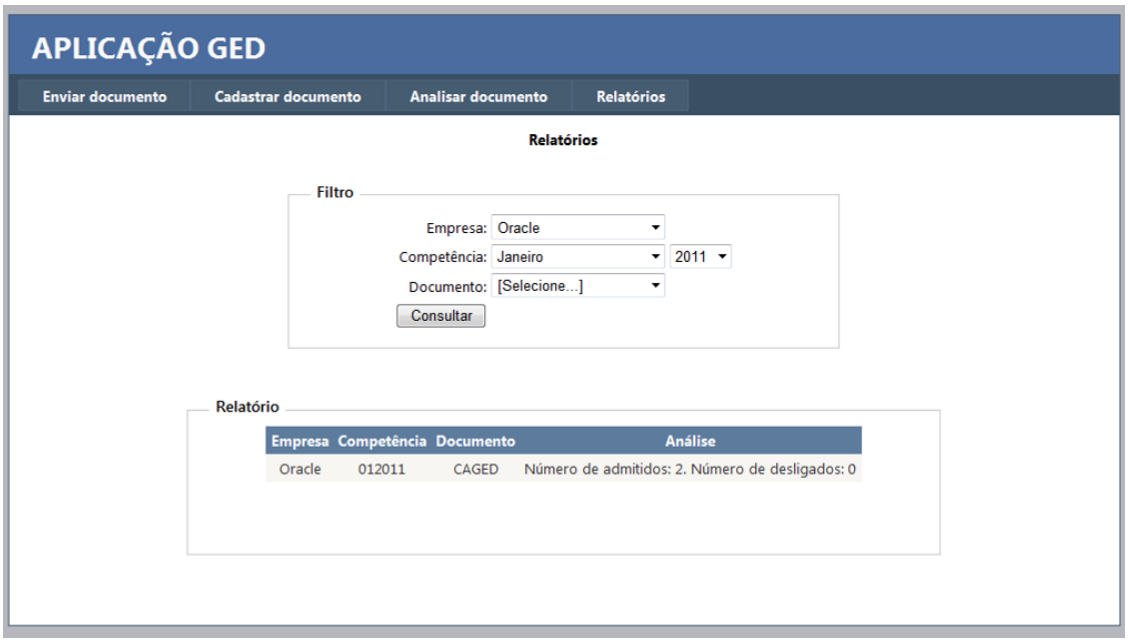


FIGURA 16: Interface *Relatório*

Fonte: Próprio autor.

5.1.4 Arquitetura

O sistema GED foi desenvolvido na arquitetura de três camadas. A camada de Interface contém as páginas WEB da aplicação, responsáveis pela interação com o usuário. A camada de negócios possui classes que realizam a lógica de negócios da aplicação. A camada de Acesso a dados é composta por classes que realizam a persistência e recuperação dos dados.

No sistema, o fluxo das mensagens ocorre sempre da camada de Interface para a camada de negócios, e da camada de negócios (BLL) para a camada de persistência (DAL).

A Figura 17 mostra a arquitetura desenvolvida. As classes foram agrupadas em pacotes lógicos, referentes a cada camada da aplicação.

As classes em destaque na camada de negócios representam os documentos criados na aplicação. A outra classe em destaque, *VinculacaoDados*, é responsável por transferir os dados do formulário dinâmico, gerado no momento da análise do documento, para seu respectivo objeto da camada de negócios. Esta classe será detalhada mais a frente. A TABELA 11 destaca os objetivos de cada classe do sistema.

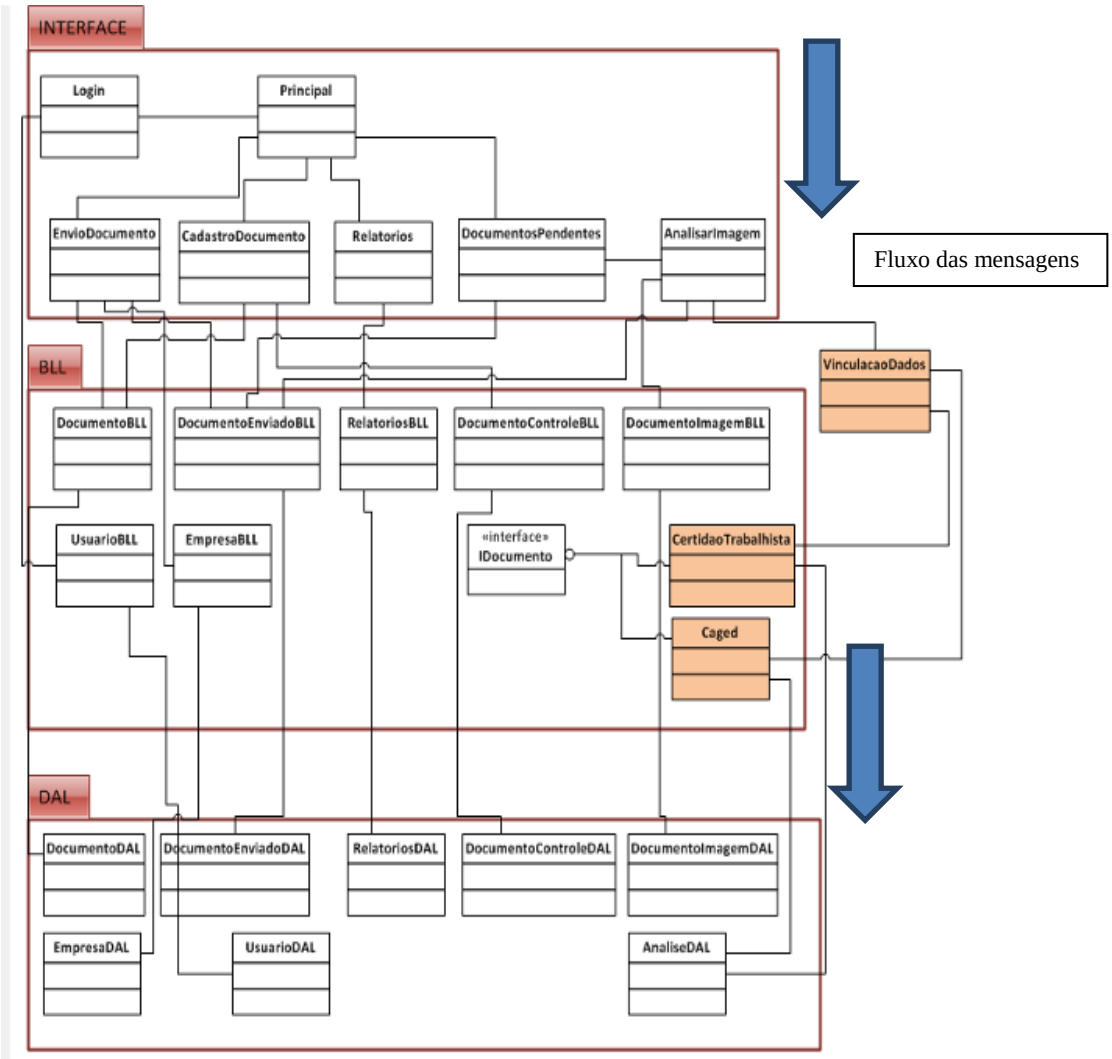


FIGURA 17: Arquitetura da aplicação GED

Fonte: Próprio autor.

Nome da classe	**Objetivo**
Login	Classe de apresentação, responsável por autenticar o usuário.
Principal	Classe de apresentação, contendo o menu principal do sistema.
Envia Documento	Classe de apresentação que permite ao usuário fazer o upload de um arquivo de imagem referente ao documento selecionado e enviá-lo para análise.
CadastroDocumento	Classe de apresentação que permite ao usuário cadastrar um novo tipo de documento, informando o nome e os campos para análise do mesmo.
DocumentosPendentes	Classe de apresentação que permite ao usuário verificar quais documentos ainda não foram analisados.
AnalisarImagem	Classe de apresentação que permite ao usuário analisar a imagem do documento e salvar os dados do mesmo.
Relatorios	Classe de apresentação que permite ao usuário visualizar os relatórios gerados pela análise.
VinculacaoDados	Classe responsável por vincular os dados informados na análise do documento para o respectivo objeto na camada de Negócios.
DocumentoBLL	Classe que possui as regras de negócios responsáveis pelos documentos cadastrados no sistema.
DocumentoEnviadoBLL	Classe que possui as regras de negócios responsáveis pelos documentos enviados para análise pelo usuário.
DocumentoControleBLL	Classe que possui as regras de negócios responsáveis pelos controles associados a cada documento cadastrado.
DocumentoImagemBLL	Classe que possui as regras de negócios responsáveis pelas imagens de cada documento enviado.
UsuarioBLL	Classe que possui as regras de negócios responsáveis pela autenticação do usuário.
EmpresaBLL	Classe que possui as regras de negócios associadas às empresas cadastradas no sistema.
DocumentoDAL	Classe de persistência responsável pela recuperação e persistência dos documentos cadastrados.
DocumentoEnviadoDAL	Classe de persistência responsável pela recuperação e persistência dos documentos enviados para análise.
DocumentoControleDAL	Classe de persistência responsável pela recuperação e persistência dos controles associados a cada documento cadastrado.

DocumentoImagemDAL	Classe de persistência responsável pela recuperação e persistência das imagens referentes aos documentos.
UsuarioDAL	Classe de persistência responsável pela recuperação dos usuários do sistema
EmpresasDAL	Classe de persistência responsável pela recuperação das empresas cadastradas no sistema.
Caged	Classe que possui as regras de negócio referentes aos documento do tipo Caged.
CertidaoTrabalhista	Classe que possui as regras de negócio referentes aos documentos do tipo Certidão Trabalhista.
IDocumento	Interface contendo a assinatura do método ValidarDocumento

TABELA 11: Objetivos das classes do sistema GED

5.2 Implementação

O sistema GED foi desenvolvido em ASP.NET, utilizando a linguagem de programação C#. A versão do arcabouço .NET utilizada foi a 3.5. O Visual Studio 2010 foi a ferramenta utilizada no desenvolvimento da aplicação.

O sistema se baseia na análise de documentos enviados pelos usuários. Cada tipo de documento possui campos específicos para a realização da análise, que são definidos no momento do cadastramento dos mesmos. Na página referente à análise do documento, esses campos são carregados, criando assim um formulário que é construído dinâmicamente, conforme o documento selecionado. O usuário Auditor então extrai as informações da imagem do documento preenchendo o formulário, e finaliza a análise. Estes dados serão processados e irão compor relatórios do sistema.

O ponto central da implementação do sistema GED é a estrutura construída para vincular os dados extraídos do formulário dinâmico para o objeto de negócios correspondente. A classe *VinculacaoDados* é a responsável por realizar tal tarefa. Ela atua como ponte entre as camadas de Interface e Negócios, recebendo os dados do formulário e atribuindo-os ao objeto responsável pelas informações. É importante destacar que as propriedades presentes no objeto de negócio devem possuir o mesmo nome dos campos do formulário dinâmico, para que a vinculação ocorra de forma

correta. Suponhamos que um formulário possua um campo denominado "*NumeroAdmitidos*". A classe de negócios correspondente deverá possuir também uma propriedade "*NumeroAdmitidos*".

Pelo fato do formulário ser construído dinamicamente, em tempo de execução, seria necessário utilizar uma estrutura que permitisse trabalhar com classes e objetos em tempo de execução, uma vez que não sabemos a priori qual tipo de documento será analisado. A biblioteca *System.Reflection* foi a utilizada para o desenvolvimento da estrutura citada, pelo fato de oferecer recursos para, em tempo de execução, instanciar objetos dinamicamente, invocar métodos, analisar a estrutura interna de classes, dentre outros recursos.

O processo de vinculação de dados se inicia com a obtenção dos dados informados no formulário dinâmico. A Listagem 5 é referente ao método *ObterValoresAnalise*(), responsável pela recuperação dos dados. A classe genérica *Dicitionary<key, value>* foi utilizada para armazenar os dados. Esta classe representa uma coleção de chaves e valores, podendo armazenar qualquer tipo de dado. No método, a classe *Dictionary* armazena, na posição chave, o ID da propriedade, e na posição de valor, o dado do campo.

```
private Dictionary<string, string> ObterValoresAnalise()
        {
            Dictionary<string, string> propriedades = null;
            try
            {

               propriedades= new Dictionary<string, string>();

                propriedades.Add("IDDocumento",
Request.QueryString["IDDocumento"].ToString());


                foreach (Control control in pnlValidacao.Controls)
                {
                    if (control is ListControl)

                        propriedades.Add(
((ListControl)control).ID,((ListControl)control).SelectedItem.Value);

                    else if (control is TextBox)

                        propriedades.Add(((TextBox)control).ID,
              ((TextBox)control).Text);
                    }
            }
            catch(Exception ex)
            {
                MensagemAlerta("Falha ao obter valores da análise: " +
ex.Message);

            }

            return propriedades;
        }
```

LISTAGEM 5: Recuperação de dados do formulário dinâmico

O passo seguinte é invocar o método *VincularPropriedades*, da classe *VinculacaoDados.* Este método é o responsável por vincular os dados do formulário para o objeto de negócios correspondente. Ele recebe como parâmetros: um objeto da classe Dictionary<>, que contém os dados do formulário, a classe a ser instanciada do objeto de negócios (recuperada da base de dados), e o protocolo do documento (gerado no momento que o usuário envia o documento, também recuperado da base de dados).

O método *VincularPropriedades* é responsável por realizar as seguintes tarefas:

1 Descobrir o tipo da classe de negócios correspondente ao documento e instanciar um objeto do mesmo. Para realizar esta tarefa, primeiramente é obtido o tipo da classe por meio do método *GetType*() da classe *System.Type*. Depois, é instanciado um objeto da classe *System.Reflection.ConstructorInfo*. Por fim, um objeto da classe é instanciado, utilizando o método *invoke()* do construtor. A Listagem 6 é referente ao trecho de código responsável por realizar as tarefas descritas.

```
            //Identifica o tipo de objeto(Classe de negócio)

            Type documentoTipo = Type.GetType("Validador.BLL.Analise." +
classeDocumento);

            //cria uma instancia da classe do documento
            ConstructorInfo construtor = documentoTipo.GetConstructor(new
Type[] { });

                //instancia o objeto da classe
            object objetoDocumento = construtor.Invoke(new object[] { });
```

LISTAGEM 6: Descoberta de tipos e instanciação de objetos

2 Descobrir as propriedades da classe e vincular os dados do formulário a elas. O objeto da classe *System.Reflection.PropertyInfo* recebe as propriedades presentes na classe de negócios, que são obtidas por meio do método *getProperties*(), presente em *System.Type().* Por meio de uma estrutura de repetição executada para cada propriedade da classe, os valores presentes na estrutura Dictionary<> passada são vinculadas as propriedades da classe, por meio do método *setValue*() da classe *System.Reflection.PropertyInfo*. Note que é utilizado o método *Convert.ChangeType*,

que irá converter o valor da variável passada na estrutura Dictionary<> para o tipo correspondente ao atributo que será vinculado. A Listagem 7 ilustra este processo:

```
                //Obtem as propriedades da classe
                PropertyInfo[] propriedadesClasse =
documentoTipo.GetProperties();

                //atribui os valores do formulario às propriedades
                foreach (PropertyInfo propriedade in propriedadesClasse)
                {

                    if (listaPropriedades.ContainsKey(propriedade.Name))

                        propriedade.SetValue(objetoDocumento,
Convert.ChangeType(listaPropriedades[propriedade.Name],
propriedade.PropertyType), null);

                }
```

LISTAGEM 7: Descoberta de propriedades e vinculação de dados.

3 Invocar o método de validação de documento da classe: O último passo do processo de vinculação de dados é a invocação do método *ValidarDocumento()* presente em cada classe de negócios referentes aos documento do sistema. O primeiro passo é verificar se a classe implementa a Interface *IDocumento*, que possui a assinatura do método *ValidarDocumento*(). Caso implemente a interface, o objeto da classe *System.Reflection.MethodInfo* recebe o método ValidarDocumento(), por meio do método getMethod(), da classe *System.Type*. Este método é então invocado, via o método *MethodInfo.Invoke*(). Caso a classe não implemente a Interface *IDocumento*, uma exceção é lançada. A Listagem 8 representa o trecho de código referente a implementação do processo descrito:

```
MemberInfo memberInterface=documentoTipo.GetInterface("IDocumento");

                if (memberInterface != null)
                {

                    //Obtem o método de validacao da classe
                    MethodInfo metodoValidacao =
documentoTipo.GetMethod("ValidarDocumento");

                    //Lista de parametros do método ValidarDocumento
                    ParameterInfo[] parametrosMetodo =
metodoValidacao.GetParameters();

                    if (parametrosMetodo.Length == 0)

                        metodoValidacao.Invoke(objetoDocumento, null);

                    else

                        metodoValidacao.Invoke(objetoDocumento, new
Object[] { pProtocolo });

                }

                else
                    throw new Exception("A classe de negócios referente
ao documento não implementa a Interface IDocumento.");
```

LISTAGEM 8: Invocação do método ValidarDocumento()

O método *ValidarDocumento*() realiza o processamento dos dados extraídos pelo usuário no momento da análise do documento. Cada documento possui suas próprias regras de negócio para realizar este processamento, uma vez que cada um possui campos diferentes a serem analisados.

Após o fim do processamento, os resultados são armazenados na base de dados, e poderão ser consultados posteriormente na tela de relatórios do sistema.

5.3 Conclusão

Nesta seção, descrevemos a aplicação GED desenvolvida, seus casos de uso, telas, classes, arquitetura e a implementação da classe *VinculacaoDados*, responsável por atribuir os valores extraídos do formulário para o objeto de negócios correspondente, por meio da reflexão computacional.

CONCLUSÃO

A reflexão computacional é um poderoso recurso fornecido por determinadas linguagens orientadas por objetos, possibilitando realizar tarefas geralmente utilizadas no momento do desenvolvimento da aplicação, em tempo de execução.

A biblioteca *System.Reflection* presente no arcabouço .NET oferece os principais recursos para realizarmos tarefas em tempo de execução: descoberta de tipos, instanciação de objetos, invocação de métodos e criação de tipos. A limitação desta biblioteca está relacionada a reflexão comportamental, que permite alterar o comportamento do sistema, por meio da interceptação de mensagem no nível base e manipulação das mesmas no meta nível. Porém, esta limitação não interferiu no desenvolvimento deste trabalho, pois apenas a reflexão estrutural foi suficiente para alcançar os objetivos propostos

A reflexão computacional aplicada no sistema GED desenvolvido foi o mecanismo encontrado para realizar a tarefa de transferência de dados entre as camadas de Interface e de Negócios da aplicação, uma vez que a aplicação possui formulários dinâmicos construídos em tempo de execução. Caso o sistema possua inúmeros tipos de documento, a criação de várias estruturas condicionais para a determinação de qual tipo de objeto de negócio deverá ser instanciado, e, quais propriedades deverão ser atribuídas de acordo com o tipo de documento analisado, se torna uma prática inviável e de difícil manutenção, aumentando consideravelmente a quantidade de linhas de código da aplicação.

O uso da reflexão computacional, apoiado pela biblioteca *System.Reflection*, foi fundamental para a realização das tarefas exigidas pela aplicação, como instanciação de objetos, descoberta de tipos e invocação dinâmica de métodos de forma automática, em tempo de execução. Esta vinculação de dados entre as camadas utilizando a reflexão computacional proporciona um ganho em produtividade, além de melhorar a manutenção dos mesmos.

Vale também ser destacado o mecanismo utilizado para captura dos dados do formulário, onde uma estrutura de repetição realiza uma varredura em todos os campos de *input* do usuário, armazenando seus identificadores e valores em um objeto da classe genérica *Dicitionary<key, value>*. Esta estrutura pode ser utilizada para a leitura de dados de qualquer formulário, o que possibilita seu reúso em outros sistemas.

O conceito utilizado não se aplica apenas a formulários dinâmicos. Em formulários padronizados, como, por exemplo, em um cadastro de usuários, o uso do mecanismo aplicado no sistema GED para captura e vinculação de dados pode ser reutilizado, contribuindo assim para a padronização de código dos aplicativos e ganho de tempo no desenvolvimento de tais estruturas, uma vez que o mesmo código pode ser aplicado em outros sistemas.

Em relação à arquitetura empregada no sistema, foi respeitada a independência entre as camadas e o fluxo de transferência de mensagens entre as mesmas, sempre da camada de Interface para a camada de negócios, e da camada de negócios para a camada de persistência. Estas características possibilitam que uma camada seja alterada sem impactar nas demais, fato que ocorre com freqüência quando novos requisitos são adicionados ou alterados em sistemas de software. Outra vantagem é a facilidade na portabilidade do sistema para outros ambientes, como mobile ou desktop, sendo

necessário alterar apenas a camada de interface do sistema, e, no caso do ambiente móbile, alterar o tipo da base de dados.

Limitações e Trabalhos Futuros

Melhorias do sistema GED: no momento do cadastro de um novo tipo de documento, são informados quais campos ele irá possuir, e estas informações são gravadas na base de dados. Porém, a classe de negócios referente ao mesmo é criada manualmente, e deve possuir propriedades referentes a cada campo cadastrado para o documento. A biblioteca *System.Reflection*, por meio da classe *Reflection.Emit*, possibilita a criação de classes em tempo de execução. Uma melhoria a ser feita no sistema seria utilizar esta biblioteca para criar os objetos de negócio referentes aos documentos no momento do cadastro dos mesmos. Fica como proposta a criação do método de validação específico para cada tipo de documento, o que irá exigir um conhecimento aprofundado da linguagem intermediária MSIL.

Desenvolvimento do sistema em outras linguagens: linguagens orientadas por objetos que suportam reflexão computacional possibilitam a implementação da solução proposta.

REFERÊNCIAS

AITKEN, Peter G. *Microsoft Basic 7.1*: A Programmer's Reference. 1. Ed. San Francisco: John Wiley & Sons Inc, 1991. 432 p.

ALLEN, Edward B. et. al. Measuring Coupling and Cohesion: An Information-Theory Approach. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS. 6.,1999, Florida. Proceedings... METRICS. [Sl]: IEEE, 1999, p. 119-127. Disponível em: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=809733. Acesso em: 02 fev 2011.

AMBLER, Scott W. *Building Object Applications That Work.*1. Ed. Cambridge: Cambridge University Press, 1998. 506 p.

BARTH, Fabricio Jailson. *Utilização de Reflexão computacional para implementação de Aspectos Não Funcionais em um Gerenciador de Arquivos Distribuídos.* 2000. 90f. Monografia (Curso de Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2000. Disponível em http://www.lume.ufrgs.br/handle/10183/2283. Acesso em: 08 Mar. 2011.

CABRAL, B. et. al. RAIL: Code Instrumentation for .NET. In: ACM SYMPOSIUM ON APPLIED COMPUTING. 5., 2005, New York. Proceedings… SAC 05: ACM, 2005, 200- 201. Disponível em: http://portal.acm.org/citation.cfm?id=1066967. Acesso em: 13 jan. 2011.

CANNOLLY, Randy. *Core Internet Application Development with ASP.NET 2.0*. 1. Ed. Massachusetts: Prentice Hall, 2007. 1049 p.

CEMBRANELLI, Felipe. *ASP.NET*: Guia do desenvolvedor. 1. Ed. São Paulo: Novatec, 2003. 256 p.

DEITEL, J. Paul. Et. Al. *Perl*: Como Programar. 1. Ed. São Paulo: Bookman, 2002. 952p.

FERREIRA, Kecia Aline Marques. *Avaliação de Conectividade em Sistemas Orientados por Objetos.* 2006. 149 f. Dissertação (Mestrado) – Universidade Federal de Minas Gerais. Departamento de Ciência da Computação, Belo Horizonte, 2006. Disponível em: http://www.bibliotecadigital.ufmg.br/dspace/bitstream/1843/RVMR-6TFPLA/1/keciaalinemarquesferreira.pdf. Acesso em: 10 jun. 2011.
FORMAN, I. R. FORMAN. *Java Reflection in Action*. 1. Ed. New York: Manning, 2004. 300 p.

GOLM, Michael, KLEINODER, Jurgen. metaXa and the future of reflection. In: OOPSLA WOKSHOP ON REFLECTIVE PROGRAMMING IN C++ AND JAVA. 5., 1998, Vancouver: OOPS, 1998. 197 – 206. Disponível em: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.9481. Acesso em: 21 abr. 2011.

HIBERNATE, *Relational Persistence for Java and .NET*. Disponível em: http://www.hibernate.org/. Acesso em: 29/04/2011

JUNIOR, Elemar. Intermediate Language - IL: Como as coisas funcionam no .NET. *Revistas .NET Magazine*. Rio de Janeiro. v.1, n.81, p. 16-23, Jan. 2011.

KICZALES, G. et. al. *The art of the metaobjects protocol*. 1. Ed. Cambridge: MIT Press, 1991. 345p.

LIBERTY, Jesse. XIE, Donald. *Programing C# 3.0*. 5. Ed. California: O´Reilly Media, 2007. 608p.

LISBOA, Maria L. B. et. al. A new trend on the development of fault-tolerant applications*:* Software meta-level architectures. *Journal of the Brazilian Computer Society* (JBCS). Rio de Janeiro. v. 4, n. 2. p. 39-47. Disponível em: http://bibliotecadigital.sbc.org.br/download.php?paper=185.
Acesso em: 20 mar. 2011.

LOPES, Juan. Reflection: Construindo uma Engine de Serialização. *Revista .NET Magazine.* Rio de Janeiro. v.1, n. 84, p.26 – 33, Mai. 2011.

MARSHALL, D. Marshall. *Programming Microsoft® Visual C# 2005*: The Language. 1. Ed. New York: Microsoft Press, 2005. 704p.

MEYER, Bertrand. *Object-oriented software construction*. 2. Ed. New York: Prentice Hall, 1997.1254 p.

MICROSOFT. *Bibilioteca System.Reflection*. Disponível em: http://msdn.microsoft.com/en-us/library/system.reflection.aspx. Acesso em: 29/01/2011

ORMANDJIEVA, O. et. al. Providing Quality Measurement for Aspect-Oriented Software Development. In: SOFTWARE ENGINEERING CONFERENCE. 12., 2005, Taipei. Proceedings… APSEC. [S1]: IEEE, 2005, p. 769-775. Disponível em: http://www.computer.org/portal/web/csdl/doi/10.1109/APSEC.2005.92. Acesso em: 02 fev 2011.

PAULA FILHO, Wilson de Pádua. *Engenharia de Software*: Fundamentos, Métodos e Padrões. Ed. 2. Rio de Janeiro: LTC, 203. 602 p.
PAULI, Guinter. POO e Multicamadas: Boas práticas de desenvolvimento com ASP.NET. *Revista .NET Magazine*. Rio de Janeiro. v.1, n.43, p. 32-33, Fev. 2007.

SANTOS, Karla Costa. *Princípios de Programação Modular*. 2006. 87 f. (Monografia) - Universidade Federal de Minas Gerais. Departamento de Ciência da Computação, Belo Horizonte, 2006.

SOUZA, Cristina V. et. al. Intercessão em Tempo de Implantação: uma Abordagem Reflexiva para a Plataforma J2EE. In: XV SIMPÓSIO BRASILEIRO DE

ENGENHARIA DE SOFTWARE, 15, 2001, Rio de Janeiro: BDBComp, 2001. 286-301. Disponível em: http://www.lbd.dcc.ufmg.br:8080/colecoes/sbes/2001/019.pdf. Acesso em 12 fev. 2011.

SULLIVAN, Gregory T. (2001). Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*. New York. V. 44. N. 10 p. 10-15. Out. 2001. Disponível em: http://portal.acm.org/citation.cfm?id=383865. Acesso em: 23 Abr. 2011.

SUN MICROSYSTEM.*Uses of Reflection.* Disponível em: http://download.oracle.com/javase/tutorial/reflect/index.html. Acesso em: 24 abr. 2011.

TATSUBORI, Michiaki. *An Extension Mechanism for the Java Language*. 1999. 64 f. Dissertação (Mestrado)- Escola de Engenharia, Universidade de Tsukuba, Tsukuba, 1991. Disponível em: http://www.csg.is.titech.ac.jp/openjava/papers/mich_thesis99.pdf Acesso em: 26 abr. 2011.

TROELSEN, Andrew. *Pro C# 2008 and the .NET 3.5 Platform*. 4. Ed. Berkeley: Apress, 2008. 1370 p.

YAMAGUTI, Marcelo. *Uma arquitetura reflexiva baseada na web para ambiente de suporte a processo*. 2002. 129 f. Monografia (Programa de Pós Graduação)- Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2002. Disponível em: http://www.lume.ufrgs.br/bitstream/handle/10183/4138/000397354.pdf?sequence=1. Acesso em: 10 jan. 2011.