Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis - IMESA

JOSÉ DAVID CAMOLEZE JÚNIOR

PROPOSTA DE UM FRAMEWORK PARA A PERSISTÊNCIA DE DADOS USANDO A TECNOLOGIA .NET

ASSIS
2012

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis - IMESA

PROPOSTA DE UM FRAMEWORK PARA A PERSISTÊNCIA DE DADOS USANDO A TECNOLOGIA .NET

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Municipal do Ensino Superior de Assis – IMESA e Fundação Educacional do Município de Assis – FEMA, como requisito para a obtenção do Certificado de Conclusão.

Orientador: Dr. Almir Rogério Camolesi
Área de Concentração: Tecnologia Adaptativa, Reflexão Computacional, Linguagem de Programação.

Assis
2012

DEDICATÓRIA

Dedico este trabalho a minha mãe que sempre foi minha base para tudo que eu construí na vida e principalmente por seu esforço para que todos os seus filhos pudessem cursar o ensino superior.

RESUMO

Esta pesquisa analisa alguns aspectos da reflexão computacional, tais como a importância da mesma em sistemas adaptativos. A reflexão permite aos sistemas que utilizam desta técnica a autoanálise e a possibilidade de modificação em tempo de execução, esses recursos surgiram na inteligência artificial e tinham como principal objetivo fazer com que sistemas conseguissem se adaptar a situações inesperadas. Além disso a reflexão permite um nível de abstração muito satisfatória e tem sido adotada pela maioria dos frameworks de persistência de dados, razão pela qual foi escolhida neste trabalho para a realização do framework com o objetivo de demonstrar as suas funcionalidades e aplicação.

Palavras-chave: Reflexão, Sistemas Adaptativos, Inteligência Artificial, tecnologia .NET.

ABSTRACT

This research analyzes some aspects of computational reflection, such as its importance in adaptive systems. Reflection allows systems that use this technique to self-analysis and the possibility of change at runtime, these features have emerged in the artificial intelligence and had as main objective to make systems were able to adapt to unexpected situations. Further reflection allows a very satisfactory level of abstraction and has been adopted by most data persistence frameworks, which is why this work was chosen for the realization of the framework with the aim of demonstrating its features and application.

Keywords: Reflection, Adaptive Systems, Artificial Intelligence, technology .NET.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

O conceito de reflexão não aparece só na computação, ele aparece em várias ciências, como filosofia, linguística e lógica. Por exemplo, "(...) no sentido de cognição humana, reflexão expressa habilidade de meditar sobre ideias, ações, sentimentos e experiências" (Correa, 1998, p.49h). Na computação a reflexão surgiu na inteligência artificial a fim de expressar, mesmo que vagamente, uma das principais funções da cognição humana. Sendo assim, as máquinas poderiam meditar sobre seu próprio código em execução abrindo inúmeras portas para a computação.

O estudo sobre uma tecnologia que pudesse adaptar-se a situações inesperadas ganhava então uma forte aliada, a reflexão computacional. Partindo do princípio em que agora uma máquina pudesse refletir sobre seu próprio código, nada impedia que além de meditar ela pudesse modificar sua execução. Sendo assim, a reflexão computacional hoje em dia é uma das técnicas mais promissoras para a inteligência artificial e foco de inúmeros estudos.

Quando a reflexão computacional alia-se com outras técnicas fornecidas pela plataforma .NET, como os tipos genéricos, ela se torna ainda mais interessante. Para demonstrar tais técnicas, será apresentado um *framework* que utiliza as tecnologias citadas.

1.1 OBJETIVOS

O projeto objetiva um estudo sobre a programação reflexiva e outras tecnologias fornecidas pela plataforma .NET que auxiliam na criação de sistemas adaptáveis e inteligentes. Ao fim da pesquisa pretende-se desenvolver um estudo de caso que utilize tais tecnologias, a fim de demonstrar algumas das funcionalidades citadas.

1.2 JUSTIFICATIVAS

O projeto justifica-se na grande necessidade de mercado, já que com a rápida evolução tecnológica, sistemas com capacidade adaptativa são cada vez mais requisitados. Ressalta-se

também a relevância na contribuição para o aumento da produção bibliográfica, já que é notável a escassez acadêmica atual acerca do tema.

1.3 PERSPECTIVAS DE CONTRIBUIÇÃO

O presente trabalho visa ampliar o leque de técnicas utilizadas pela tecnologia adaptativa e pela inteligência artificial, utilizando as técnicas que serão apresentadas neste trabalho. Além disso, contribuir para o aumento da produção bibliográfica para pesquisadores que desejam realizar estudos sobre as funcionalidades da reflexão computacional.

1.4 METODOLOGIA DE PESQUISA

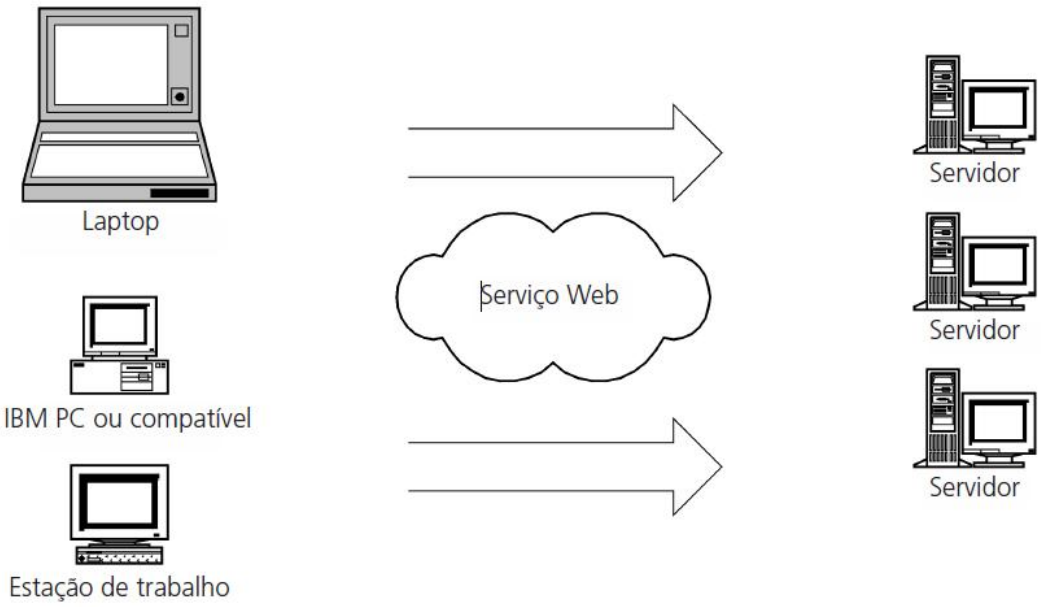
A metodologia de pesquisa adotada será a experimental. Inicialmente será realizada a leitura de artigos e livros relacionados à reflexão computacional e sobre outras tecnologias da plataforma .NET. Posteriormente, será desenvolvido um projeto que demonstra o uso das tecnologias estudadas.

2. PLATAFORMA .NET

Escolher a linguagem que será usada para a criação de um sistema sempre foi uma questão muito discutida por qualquer empresa. Normalmente essas empresas optam por uma linguagem de acordo com a plataforma que será usada, pela cultura da empresa, pelos preços e pelo próprio gênero do sistema. Frequentemente desenvolvedores precisam utilizar mais de uma linguagem de programação para o mesmo sistema, devido a particularidade que muitas vezes uma linguagem apresenta. A Microsoft .NET lida perfeitamente com este tipo de problema.

A plataforma .NET permite o desenvolvimento utilizando as linguagens de sua preferência, tais como: C#, Visual Basic, Perl, C++ e etc. Além de ser uma plataforma de múltiplas linguagens ela permite a perfeita integração entre componentes diferentes escritos em linguagens diferentes.

O principal foco da plataforma é o desenvolvimento de Serviços WEB, ou mais conhecido como WEB Services. Por meio de WEB Services sistemas desenvolvidos em quaisquer linguagens e rodando em quaisquer Sistemas Operacionais (SO) podem trocar informações. Hoje em dia essa perspectiva de WEB Services é foco de pesquisas e desenvolvimento de grandes corporações.



(LIMA; REIS, 2002)

Segundo Carlos Vamberto o .NET é um "ingrediente" sempre presente em toda a linha de produtos da Microsoft, oferecendo a capacidade de desenvolver, implementar, gerenciar e usar soluções conectadas através de Web Services XML, de maneira rápida, barata e segura. Essas soluções permitem uma integração mais ágil entre os negócios e o acesso rápido a informações a qualquer hora, em qualquer lugar e em qualquer dispositivo.

2.1 GENERICS

Os genéricos são recursos muito poderosos que foram adicionados a versão 2.0 do C#. Eles permitem a definição de estruturas de dados sem se comprometer aos reais tipos de dados que tal estrutura pode pertencer. Com isso os códigos ganham desempenho e qualidade, pois funções que tenham um mesmo objetivo, porém que trabalham com diferentes tipos de dados podem ser escritas apenas uma vez.

Usando um genérico podemos implementar uma função que tenha uma finalidade comum para os mesmos objetos apenas uma vez. Por exemplo, podemos definir uma função que declare um parâmetro genérico e use essa variável como qualquer tipo. Para que

possamos trabalhar com esses tipos em alguma classe, temos que adicionar o <T> colocando um parâmetro do tipo genérico na própria classe, por exemplo:

```
namespace FrameworkDB
{
    public class GenericDAL<T>
    {
```

Figura 1 Exemplo de Generics na Classe

Quando uma classe é parametrizada com o tipo genérico, ela passa a ter a capacidade de trabalhar com os mesmos em suas funções, como na figura 2.

```
#region Protected Execute Procedures Methods
    protected void Add(T pObj)
    {
        this.Insert(pObj);
    }
    protected void Del(T pObj)
    {
        this.Delete(pObj);
    }

    protected void Upd(T pObj)
    {
        this.Update(pObj);
    }
#endregion
```

Figura 2 Exemplo Funções Generics

Para utilizarmos essas funções é preciso que na hora de instanciar a classe, para a criação do objeto, o programador parametrize a classe com o tipo que substituirá o objeto genérico. Outra forma, a utilizada pelo projeto desenvolvido, é parametrizar com o tipo que substituirá o tipo genérico na hora da herança, por exemplo, o projeto desenvolvido conta com uma camada intermediária entre a camada de persistência e a camada de apresentação. As classes que envolvem a camada intermediária, chamada de *business*, herdam uma classe que foi parametrizada com um tipo genérico, e na hora da herança o tipo genérico já é definido pelo programador, como mostra a Figura 3. A partir disso, todas as funções definidas como genéricas agora passarão a trabalhar com o tipo definido.

```
public class FaculdadeBO : GenericDAL<Faculdade>
{
    public void ADD(Faculdade vo)
    {
        using (base.TransactionScope())
        {
            this.Add(vo);
        }
    }
```

Figura 3 Definindo o Tipo Genérico

2.2 ATTRIBUTES

Um atributo é um elemento que permite adicionar metadados ao código objeto (*assembly*) do programa. É uma classe especial para armazenamento de informações sobre algum elemento no seu código. O elemento em que um *attribute* é aplicado é chamado de alvo [*target*], esses alvos podem ser funções, propriedades, classes e etc.

O objetivo de um *attribute* é fazer o compilador anexar ao *assembly* metadados sobre o programa. Para aplicar um *attribute* o programador deve adicioná-lo entre colchetes exatamente acima do construtor.

```
[AttTableDB("Faculdade")] //Attribute
public class Faculdade
{
```

Figura 4 Exemplo Attribute

Além de utilizar atributos prontos definidos pela plataforma .Net, o programador pode criar novos de acordo com as próprias necessidades e definindo que tipos de metadados iremos guardar sobre certo alvo.

```
namespace Attributes
{
    public enum FieldType
    {
        Default = 0,
        PrimaryKey = 1,
        PrimaryKeyAI = 2
    }

    [AttributeUsage(AttributeTargets.Property)]
    public class AttFieldDB : Attribute
    {
        private string _fieldName;
        private FieldType _fieldType;
        private DbType _dbType;

        public string FieldName
        {
            get { return _fieldName; }
            set { _fieldName = value; }
        }


        public FieldType FieldType
        {
            get { return _fieldType; }
            set { _fieldType = value; }
        }
```

Figura 5 Exemplo de Criação de Attribute

2.3 REFLECTION

A Programação reflexiva pode ser definida pela habilidade de gerar fragmentos de meta-código e integrá-los a sua execução, modificar sua própria estrutura e se autoanalisar. Contudo, a ênfase da programação reflexiva é a modificação ou a análise dinâmica, ou seja, todas as habilidades citadas anteriormente são definidas em tempo de execução. “Reflection in a programming language can be used to observe and dynamically modify or change the program execution at runtime” (Naim et al., p.43).

A classe abstrata *System.Type* é a mais importante para a reflexão computacional, ela é a responsável por conter todas as características de um tipo. Como a classe é abstrata não é possível instâncias diretas, porém o *Reflection* permite criar instâncias de um Type em tempo de execução. Com um objeto do tipo *Type* pode-se obter todos os dados de métodos, propriedades, campos do tipo e eventos.

O *reflection* foi utilizado na maior parte das funções de persistência do projeto, tudo graças as suas funções de autoanalise. Com tais funções é possível em tempo de execução saber qual tipo do objeto que a função está trabalhando, listar todos os seus campos e respectivos valores. A Figura 6 mostra uma das funções de persistência utilizando métodos como *GetType e GetProperties*, responsáveis por listar todas as informações de um tipo.

```
        StringBuilder sb = new StringBuilder();
        sb.Append("UPDATE " + obj.GetType().Name + " SET ");

try
{
    foreach (PropertyInfo pInfo in obj.GetType().GetProperties())
    {
                atributosField = pInfo.GetCustomAttributes(false);
```

Figura 6 Exemplo de Autoanalise com Reflection

3. PROPOSTA DE TRABALHO

O projeto desenvolvido para a demonstração do uso das tecnologias apresentadas no trabalho foi um framework de persistência. Em uma visão geral, o framework tem o papel de persistir dados genéricos, ou seja, ele consegue lidar com qualquer tipo de informação que possa ser gravada em um banco de dados sem que essas informações tenham sido previamente tratadas, e gravá-las em qualquer banco de dados.

A imagem abaixo retrata uma visão geral do sistema. Primeiramente, o programador que utilizará o *framework* deve mapear as classes que serão persistidas, para tal mapeamento foram disponibilizados dois *attributes*. O *framework* foi dividido em duas partes, a primeira é composta pela biblioteca de conexão, responsável por identificar o banco de dados utilizado pelo sistema e retornar todos os objetos de conexão que podem atuar sobre o mesmo. A segunda parte é composta pela biblioteca de persistência, que tem a função de analisar as classes mapeadas, gerar o código SQL e utilizar os objetos de conexão retornados pela biblioteca de conexão para executar os mesmos.

Em suma, o programador não terá nenhum trabalho com funções de persistência nem tampouco com as conexões com o banco de dados. A única programação que deverá ser feita é o mapeamento das classes e uma camada que herde as funcionalidades do framework, chamada neste projeto de camada intermediária.

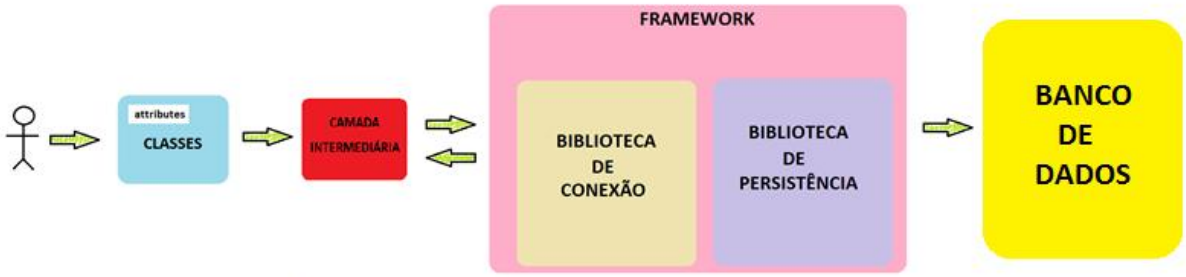


Figura 7 Visão Geral do Framework

3.1 MAPEAMENTOS DAS CLASSES

Em todos os sistemas de persistência as classes que serão persistidas necessitam passar por algum tipo de mapeamento para que o *framework* saiba que determinada tabela é relacionada à determinada classe. Neste projeto foi abordada uma técnica que permite adicionar metadados a quaisquer tipos de construtores, propiciando aos programadores que utilizarem o *framework* adicionar junto aos campos da classe o nome do campo referido na tabela. Neste projeto foram utilizados *attributes* próprios para adicionar metadados às classes.

Na classe Faculdade, Figura 8, foram usados dois *attributes* próprios para o mapeamento da mesma. Primeiramente utilizando o *attribute AttTableDB*, é adicionado junto ao construtor da classe um metadado com o nome da tabela referente, assim em tempo de execução pode-se identificar que quando o sistema for persistir os elementos desta classe ele deverá usar a tabela Faculdade.

Seguindo este pensamento, utiliza-se o *attribute AttFieldDB* para adicionar junto aos construtores dos métodos assessores (*getters*) e modificadores (*setters*) de cada atributo da classe os nomes referentes aos campos da tabela. Além do nome esse *attribute* permite adicionar o tipo do campo, por exemplo, *FieldType. PrimaryKey.* Essa diferenciação dos campos é de extrema importância no processo de criação dos códigos SQL.

Com todos os metadados corretamente inseridos a classe é mapeada. Quando o programa é executado pode-se extraí-los de dentro do *assembly* e trabalhar com essas informações, no caso do *framework* esses dados serão usados para a criação dos códigos SQL.

```
namespace DAO
{
    [AttTableDB("Faculdade")]
    public class Faculdade
    {
        int id;
        String descricao;

        [AttFieldDB("ID", FieldType.PrimaryKeyAI)]
        public int Id
        {
            get { return id; }
            set { id = value; }
        }

        [AttFieldDB("Descricao")]
        public String Descricao
        {
            get { return descricao; }
            set { descricao = value; }
        }
    }
}
```

Figura 8 Classe Faculdade Mapeada

3.2 BASE DE DADOS GENÉRICA

Um dos grandes méritos do projeto é a capacidade do mesmo de trabalhar com tipos genéricos em vários módulos. Um dos módulos é o que tem o papel de identificar qual banco de dados o sistema usa e retornar objetos de conexão compatíveis a ele.

Este módulo foi retirado de um artigo do DevMedia, escrito por Leandro Ribeiro. Em seu artigo Leandro desenvolve uma ferramenta que centraliza em apenas uma função a leitura e execução de qualquer *procedure,* independente do banco de dados.

Para que esse módulo funcione, o programador deve trabalhar com definições de conexões (*connections strings*) definidas nos arquivos de configuração. Essa é única

configuração que de fato envolve o banco de dados elaborado pelo programador, apontar o caminho do mesmo para que o *framework* possa identifica-lo e fazer os devidos tratamentos.

```
<appSettings>
  <add key="current.connection" value="sqlServerConn"/>
 </appSettings>

 <connectionStrings>
  <add name="sqlServerConn" connectionString="Data
        Source=SRVWEB01\PRODUCAO;User
        Id=sa;password=jc6000;database=ReflectionTestes;"
        providerName="System.Data.SqlClient"/>
 </connectionStrings>
```

Figura 9 Arquivo de Configuração

O *framework* recupera a *connection string* configurada pelo programador, faz a conexão com o banco de dados e retorna os objetos de conexão que futuramente serão usados para qualquer função de persistência.

Como dito anteriormente, um dos grandes méritos do *framework* é trabalhar com tipos genéricos em vários lugares, e para que se consiga trabalhar com qualquer banco de dados não se pode trabalhar com objetos de conexão tipados. Para resolver essa incógnita foram feitos o uso das interfaces *IDbConnection* e *IDbDataAdapter*, que permite trabalhar genericamente com todos os tipos de conexão, já que todos os objetos de conexão implementam as interfaces citadas.

As classes que têm o papel de tratar as configurações relacionadas ao banco de dados são a *DBManager* e a *Factory*. A classe *DBManager* apenas tem a função de popular os objetos que serão usados pelas funções de persistência, para popular esses objetos ela usa as funções que a classe *Factory* abriga. A responsável por retornar os objetos de conexão é a classe *Factory*, ela contém as funções que retornam os objetos que implementam as interfaces *IDbConnection* e *IdbDataAdapter*.

A figura 10 contém a função que recebe o nome da *connection string* e retorna o objeto corretamente tipado, de acordo com o banco. Para que isso ocorra a função compara

uma propriedade de um objeto *ConnectionStringSettings* chamada de *ProviderName* com tipos de *providers* que o *framework* aceita. Quando ele identifica qual tipo de *ProviderName* está sendo usado, o *framework* se torna capaz de retornar um objeto de conexão que corresponde ao banco de dados usado.

```
private IDbConnection GetConn(string ConnNameConfig)
{
    IDbConnection retorno = null;
    ConnectionStringSettingsCollection connections = ConfigurationManager.ConnectionStrings;
    if (connections != null && connections.Count > 0)
    {
        foreach (ConnectionStringSettings setting in connections)
        {
            if (setting.Name.ToLower().Equals(ConnNameConfig.ToLower()))
            {
                switch (setting.ProviderName.ToLower())
                {
                    case "system.data.sqlclient":
                        retorno = new SqlConnection(ConfigurationManager.ConnectionStrings[ConnNameConfig].ConnectionString);
                        break;
                    case "system.data.odbc":
                        retorno = new OdbcConnection(ConfigurationManager.ConnectionStrings[ConnNameConfig].ConnectionString);
                        break;
                    case "system.data.oracleclient":
                        retorno = new OracleConnection(ConfigurationManager.ConnectionStrings[ConnNameConfig].ConnectionString);
                        break;
                }
            }
        }
        if (retorno == null)
        {
            throw new Exception("A chave 'current.connection' não possui o nome de uma conexão existente no web config");
        }
    }
    else
    {
        throw new Exception("Root de conectionStrings não encontrada no arquivo de configuração");
    }

    return retorno;
}
```

Figura 10 Função que retorna o objeto do tipo Connection

O mesmo é feito para recuperar os objetos do tipo *DataAdapter*, na Figura 11, porém o retorno da função é um objeto do tipo *DataAdapter*.

```
private IDbDataAdapter GetCurrAdapter(string ConnNameConfig)
{
    IDbDataAdapter retorno = null;
    ConnectionStringSettingsCollection connections = ConfigurationManager.ConnectionStrings;

    if (connections != null && connections.Count > 0)
    {
        foreach (ConnectionStringSettings setting in connections)
        {
            if (setting.Name.ToLower().Equals(ConnNameConfig.ToLower()))
            {
                switch (setting.ProviderName.ToLower())
                {
                    case "system.data.sqlclient":
                        retorno = new SqlDataAdapter();
                        break;
                    case "system.data.odbc":
                        retorno = new OdbcDataAdapter();
                        break;
                    case "system.data.oracleclient":
                        retorno = new OracleDataAdapter();
                        break;
                }
            }
        }
        if (retorno == null)
        {
            throw new Exception("A chave 'current.connection' não possui o nome de uma conexão válida");
        }
    }
    else
    {
        throw new Exception("Root de conectionStrings não encontrada no arquivo de configuração");
    }
    return retorno;
`
```

Figura 11 Função que retorna objeto do tipo DataAdapter

3.3 A CLASSE DE PERSISTÊNCIA

A responsável por todas as funções que geram os códigos SQL é a classe *GenericDAL*. Esta classe é a responsável por ler qualquer tipo de objeto previamente mapeado em tempo de execução e gerar determinado código SQL. Para isso, foram usados os conceitos de tipos genéricos juntamente com o *reflection*.

Os tipos genéricos são usados para que todos os objetos que serão persistidos possam usar as mesmas funções que farão a inclusão, exclusão e etc. Portanto, a classe *GenericDAL* foi parametrizada com o tipo <T>, deste modo todas as funções que trabalharem com o tipo genérico T podem assumir que um objeto deste tipo pode ser de qualquer classe, por exemplo, para a mesma função eu posso passar por parâmetro um objeto do tipo Pessoa ou do tipo Faculdade, sem que esta função tenha sido previamente configurada para receber tais tipos.

```
public class GenericDAL<T>
```

Figura 12 Classe Genérica GenericDAL

```
private void Insert(T obj)
```

Figura 13 Função Genérica Insert

O *reflection* é usado na definição de todas as funções que geram código, a técnica usada para esse framework foi a de autoanálise. Quando o sistema é executado e um evento que chame alguma função de persistência é disparado, o *reflection* extrai de dentro do *assembly* todos os metadados necessários para montar a SQL correspondente.

De todos os dados extraídos é possível identificar que tipo de objeto foi passado para a função que recebe dados genéricos, após identificar o objeto extraem-se todos os dados que foram passados pelo mapeamento feito anteriormente para que seja possível montar os parâmetros de inclusão, e para saber qual a tabela do banco de dados que será persistida. Além dos metadados adicionados aos campos do objeto, extrai-se também o nome do campo do objeto e o valor que ele representa.

```
private void Insert(T obj)
{
    List<string> lst = new List<string>();
    String st = " VALUES ("; //Primeira String de concatenação
    AttFieldDB attField = null;

    IDbCommand command = null; //Interfaces genéricas
    IDbDataParameter param = null; //Interfaces genéricas

    command = this._dbManager.Conn.CreateCommand();
    command.Transaction = this._dbManager.Trans; //Obtém transação

    object[] atributosField = null;

    StringBuilder sb = new StringBuilder();
    sb.Append("INSERT INTO " + obj.GetType().Name + " ("); //Segunda String de concatenação

    try
    {
        foreach (PropertyInfo pInfo in obj.GetType().GetProperties()) //Trabalhando com todos os campos do objeto
        {
            atributosField = pInfo.GetCustomAttributes(false);
            foreach (object att in atributosField) //Trabalhando com todos os attributes de cada campo
            {
                if (att is AttFieldDB)
                {
                    attField = (AttFieldDB)att;
                    if (attField.FieldType != FieldType.PrimaryKeyAI)
                        sb.Append(attField.FieldName + ", "); //Concatenando na primeira String os nomes dos campos
                }                                             // que receberão os valores
            }
```

Figura 14 Função de Inclusão Parte 1

A função *Insert* (Figura 15) trabalha com duas strings de modo geral. Primeiramente, utilizando as técnicas reflexivas, foi desenvolvido um laço de repetição para percorrer todos os campos que o objeto passado por parâmetro possua, na sequencia o outro laço de repetição é utilizado, dessa vez para percorrer todos os *attributes* que certo campo possua.

A primeira *string* é montada dentro do segundo laço, o que percorre os *attributes* de determinado campo. A *string* é formada com o nome dos campos da tabela que receberão os valores do objeto, por exemplo, “VALUES (ID, Descricao)”.

```
            if (attField.FieldType != FieldType.PrimaryKeyAI && attField != null)
            {
                param = command.CreateParameter(); //Criando os parâmetros sql
                param.ParameterName = "@" + pInfo.Name;
                param.Value = pInfo.GetValue(obj, null);
                command.Parameters.Add(param);
                st = st + "@" + pInfo.Name + ", "; //Concatenado na segunda String o nome de cada parametro criado
            }
        }

        st = st.Substring(0, st.Length - 2);
        String result = sb.ToString().Substring(0, sb.ToString().Length - 2) + ")" + " " + st + ")"; // Junta as 2 Strings

        command.CommandText = result;
        command.ExecuteNonQuery(); //Executa a SQL
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Figura 15 Função de Inclusão Parte 2

A segunda *string* é formada dentro do primeiro laço, o laço que percorre os campos do objeto. Ela é formada com o nome dos parâmetros criados para a passagem de valores, por isso os parâmetros foram criados antes de tudo. Os parâmetros recebem o nome do campo e o valor correspondente. Após a criação dos parâmetros a segunda *string* será formada, por exemplo, “INSERT INTO Faculdade (@ID, @DESC)”.

As duas *strings* então são concatenadas e em seguida executadas. Para a criação dos parâmetros e dos comandos foram usados os objetos *IDbDataParameter* e *IdbCommand* respectivamente, já que ambos podem trabalhar com qualquer tipo de base de dados.

4. CONCLUSÃO

A programação reflexiva é muito promissora para os sistemas que envolvem a Inteligência Artificial, pois além de proporcionar técnicas que realizam a autoanálise e a automodificação, ela permite trabalhar com um nível de abstração muito alto, como visto no presente trabalho. Aliando a reflexão computacional com o uso de tipos genéricos os sistemas ficam cada vez mais inteligentes e adaptáveis, pois eles passam a receber qualquer tipo de informação e a partir delas tomar alguma decisão.

As técnicas apresentadas neste trabalho ampliam a visão da inteligência artificial (IA), tendo em vista que a IA e a tecnologia adaptativa caminhem juntas. Técnicas como essa merecem atenção especial e devem se tornar alvos maiores de pesquisadores da área.

Utilizando as técnicas reflexivas e genéricas analisadas no framework desenvolvido, pretende-se desenvolver futuramente um estudo de caso sobre um gerador de diagramas de sequência. As ideias surgiram durante o estudo das técnicas de autoanálise do *reflection*, com elas pode-se mapear o sistema inteiro, ou seja, é possível saber qual função o programa está executando e como ou por quem essa função foi chamada. Portanto, se fosse possível colocar todo esse mapeamento nos padrões do diagrama de sequência e depois imprimi-lo, o gerador de diagramas estaria criado.

3. Referências

CORRÊA, SAND LUZ. **Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos**. 1998. 116p. Dissertação(Mestrado) – UNICAMP, SP, Campinas.

LIMA, EDWIN; REIS, EUGÊNIO. **C# .NET – Guia do Desenvolvedor**. Editora Campus LTDA. 2002.

NAIM, RANA; NIZAM, MOHAMMAD FAHIM; NOUREDDINE, JALAL; HANAMASAGAR, SHEETAL, **Comparative Studies of 10 Programming Languages within 10 Diverse Criteria.** 126p. Concordia University Montreal, Quebec, Canada.

RIBEIRO, Leandro. **Utilizando System.Reflection e System.Attributes para a construção de uma Ferramenta ORM.** Disponível em: <http://www.devmedia.com.br/utilizando-system-reflection-e-system-attributes-para-a-construcao-de-uma-ferramenta-orm-parte-2/21921>. Acesso em: 5 ago. 2011.

SOLIS, DANIEL; **Illustrated C# 2010,** APRESS, 2010.

STEMPLE, DAVID; GRAHAM, KIRBY; MORRISON RON, **Linguistic Reflection in Java.** 25p. Department of Computer Science, University of Massachusetts, Amherst, USA; School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Scotland.