Aplicação da Tecnologia Adaptativa e Reflexão Computacional por meio da programação reflexiva em Java

D. A. Passareli, A. R. Camolesi, D. M. R. Barros

diepassa@bol.com.br, camolesi@femanet.com.br, diomara@femanet.com.br
Coordenadoria de Informática, Fundação Educacional do Município de Assis, Brasil

***Resumo* — Este trabalho apresenta conceitos de Tecnologia Adaptativa e Reflexão Computacional. Baseado nessas tecnologias e utilizando programação reflexiva em Java, foi desenvolvido um estudo de caso, que se trata de uma tela para cálculo da mensalidade de um plano de saúde, onde, as vantagens em nível de desenvolvedor são muito grandes, como facilidade na manutenção do código fonte e otimização do trabalho. Para ilustrar o funcionamento da aplicação, foi desenvolvido também um Autômato de Estados Finitos Adaptativo.**

***Abstract* — This paper presents concepts of Adaptive Technology and Computational Reflection. Based on these technologies and using reflective programming in Java, a case study was developed, that it is a canvas for calculating the monthly payment of a health plan, where the benefits at the level of developer are very large, as was developed in ease of maintenance source code and optimization work. To illustrate the operation of the application, was also developing a Finite Automaton for Adaptive States.**

***Palavras Chave* — Tecnologia Adaptativa, Reflexão Computacional, Programação Reflexiva, Java.**

I. INTRODUÇÃO

É comum encontrar no meio comercial, sistemas que não se modificam, em comportamento e estrutura, para solucionar um problema ou em situações inesperadas.

A Tecnologia Adaptativa deu origem à ideia de que qualquer dispositivo que possua um conjunto fixo e finito de regras para sua operação, pode variar de acordo com outro nível de regras, denominado *ações adaptativas*, que agem sobre o conjunto de regras original através das ações de inserção, remoção e consulta das mesmas, podendo assim alterar sua estrutura interna durante seu funcionamento [1].

O emprego da Tecnologia Adaptativa em sistemas comerciais, que em sua maioria apresentam estruturas fixas, é possível graças a técnicas de programação que permitem modificar suas estruturas em tempo de execução. Uma dessas técnicas é a Reflexão Computacional, que torna possível ao programador acessar as políticas de controle de execução [2].

II. DISPOSITIVOS ADAPTATIVOS

A teoria dos dispositivos baseados em regras adaptativos tem como base a ideia de que dispositivos com maior poder de expressão podem ser obtidos a partir de uma progressão de um dispositivo mais simples [1].

Um simples dispositivo guiado por regras inicia seu funcionamento com certa configuração e ao longo de seu funcionamento, de acordo com os estímulos de entrada, alterna entre as configurações baseadas nas regras existentes até que nenhuma regra possa ser aplicada ou até que os estímulos de entrada terminem [1].

Já um dispositivo adaptativo, pode alterar seu conjunto de regras em função dos estímulos de entrada, baseado em outro nível de regras. Esse segundo nível de regras, chamado de *camada adaptativa*, age sobre o nível de regras original, chamado de *camada subjacente*, transformando assim um dispositivo qualquer guiado por regras em um dispositivo adaptativo [1], como pode ser observado na Figura 1.

A camada adaptativa é exclusivamente dedicada à descrição dos fenômenos de automodificação que devem ocorrer no dispositivo adaptativo que se deseja construir [3].

Figura 1: Estrutura Geral de um dispositivo adaptativo.

Um dos dispositivos abstratos que incorporam de forma mais natural esse recurso é o autômato, entendido como qualquer dispositivo com estados cuja operação seja definida por um conjunto de transições entre esses estados [4].

Durante a execução das suas transições, o autômato pode sofrer mudanças em sua topologia, tanto de ampliação ou de redução em seu conjunto de estados e transições iniciais à medida que vai reconhecendo a cadeia de entrada [5].

Um autômato adaptativo possui como camada subjacente um autômato de pilhas estruturado e, como camada adaptativa, as ações implementadas por funções adaptativas. Essas funções determinam as modificações que devem ser realizadas na camada subjacente quando uma ação adaptativa é chamada [1].

III. REFLEXÃO COMPUTACIONAL

O fundamento de Reflexão Computacional originou-se em lógica matemática e, recentemente, mecanismos de alto nível o tornam um aliado na adição de características operacionais ou não funcionais a módulos já existentes [6].

A reflexão pode ser definida como qualquer ação executada por um sistema computacional sobre si próprio, em outras palavras, é a capacidade de um programa reconhecer detalhes internos em tempo de execução que não estavam disponíveis no momento da compilação [7].

Quando um programa reflexivo entra em execução, ele considera suas próprias condições e informações contextuais. Deste modo, um programa reflexivo tem a habilidade de "pensar" sobre o que está acontecendo e se alterar dependendo das circunstâncias [8].

O termo reflexão tem dois significados distintos: introspecção, que se refere ao ato de examinar a si próprio, e redirecionamento da luz. Na Ciência da Computação, reflexão computacional denota a capacidade de um sistema examinar sua própria estrutura e estado (relacionado à introspecção) e o poder de fazer alterações no seu comportamento através do redirecionamento [7]. A representação das informações manipuláveis de um sistema sobre si próprio é chamada de metainformação [9].

Para que o mecanismo de reflexão seja implementado de forma flexível é necessário definir uma arquitetura reflexiva [10]. A arquitetura reflexiva compõe-se de dois níveis: *metanível* e *nível base*. No metanível se encontram as estruturas de dados e as ações que são executadas sobre o sistema objeto que está presente no nível base [8].

O metanível é explorado para expressar propriedades não funcionais do sistema, de forma independente do domínio da aplicação. Essas propriedades não funcionais, ao contrário das funcionais, não apresentam funções a serem realizadas pelo software, mas comportamentos e restrições que este software deve satisfazer [6].

A Figura 2 representa o processo de reflexão em um sistema computacional que pode ser dividido em vários níveis, onde o usuário envia uma mensagem ao sistema computacional e ela é tratada pelo nível funcional, que é responsável por executar corretamente a tarefa. Assim, o nível não funcional realiza a tarefa de gerenciar o funcionamento do nível funcional [6].

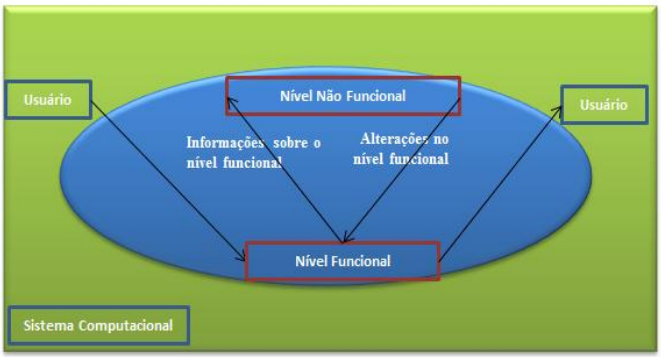

Figura 2: Visualização genérica de um sistema computacional reflexivo.

Segundo Wu (1997 apud BARTH, 2000, p. 20), o conceito básico sobre reflexão computacional está em separar as funcionalidades básicas das funcionalidades não básicas através de níveis arquiteturais, onde as funcionalidades básicas devem ser efetuadas pelos objetos da aplicação e as não básicas pelos metaobjetos. As capacidades não funcionais são adicionadas aos objetos da aplicação através de seus metaobjetos específicos e o objeto base pode ser alterado em estrutura e comportamento em tempo de execução. Metaobjetos são instâncias de uma classe pré-definida ou de uma subclasse da classe pré-definida.

Atribuir a um sistema tal capacidade significa dar-lhe flexibilidade para se adaptar dinamicamente, em estrutura e comportamento, favorecendo a reutilização (independente das classes do programa de nível base e do programa de metanível) e proporcionando adaptatividade [6].

No nível base são encontradas as classes e objetos pertencentes ao sistema objeto, e tem a funcionalidade de resolver problemas e retornar informações sobre o domínio externo, enquanto o nível reflexivo (metanível), que possui metaclasses e metaobjetos, resolve os problemas do nível base e retorna informações sobre a computação do objeto, podendo adicionar funcionalidades extras a ele [10].

A arquitetura reflexiva admite diversos metaníveis, caracterizando uma torre de reflexão, onde cada nível da torre estabelece um domínio D_i, tornando-se domínio base do domínio D_{i+1} [6]. Essa torre de reflexão pode ser analisada na Figura 3.

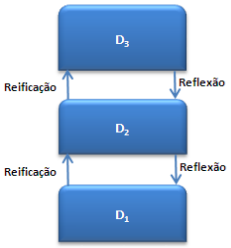

Figura 3: Torre reflexiva.

Na Figura 3, D_1 é o nível base da aplicação, D_2 é o metanível do D_1 e o nível base do D_3, e D_3 é o metanível do D_2 [6]. È possível dizer que esse é um exemplo de um adaptativo do adaptativo.

Segundo Souza (2001 apud SOUSA, 2002, p. 15), a reificação (materialização) é o ato de converter o que estava previamente implícito em algo explicitamente formulado, que é então disponibilizado para manipulação conceitual, lógica ou computacional.

É através do processo de reificação que o metanível obtém as informações estruturais internas dos objetos do nível base, tais como métodos e atributos [8].

IV. TECNOLOGIA JAVA

Java é uma plataforma e uma linguagem de programação de alto nível orientada a objetos, segura e independente de plataforma, lançada pela Sun Microsystems em 1995, mas que se originou com um projeto em 1991 sendo baseado nas

linguagens de programação C/C++.

A linguagem Java é compilada e diferentemente de muitas linguagens, seu código é primeiramente compilado para *bytecode* para depois ser executado pela JVM (*Java Virtual Machine*). Bytecode é uma espécie de código assembler para a JVM. Este código é otimizado pela JVM, que o interpreta, gerando e passando ao hardware em que esta instalada, os comandos necessários.

A maioria dos programas é escrita para trabalhar com dados, em geral para ler, escrever, manipular e exibir dados. Para alguns tipos de programas os dados a serem manipulados não são números ou textos, mas são as informações sobre programas e tipos de programa. Essas informações são conhecidas como metadados, que permitem a um assembly e os tipos dentro do assembly se autodescreverem [11].

Um programa pode olhar para os metadados enquanto ele está em execução e, isso é chamado de *reflection* (reflexão) [11]. A ênfase da programação reflexiva é a modificação ou a análise dinâmica, podendo modificar sua estrutura e praticar a autoanálise em tempo de execução.

A programação reflexiva em Java é possível graças à API (*Application Programming Interface*) *Reflection*, que fornece novos mecanismos para auxiliar no desenvolvimento de software.

A metaprogramação possui como principais vantagens a criação de aplicativos mais dinâmicos e a consequente redução do código fonte implementado. Contudo, como principais desvantagens ela apresenta a exigência de um maior nível de atenção e um domínio mais avançado de lógica de programação, além da dependência de linguagem [12].

A linguagem Java possui um pacote de reflexão, que suporta introspecção sobre classes e objetos atuais na JVM, permitindo manipular classes, interfaces, atributos e métodos em tempo de execução, a *java.lang.reflect*.

Com a programação reflexiva em Java é possível obter várias informações sobre o código fonte durante o tempo de execução, tais como a classe de um objeto, o pacote de uma classe, os atributos e métodos de uma classe e etc. É possível também criar uma instância de uma classe dinamicamente e obter e alterar os valores de atributos de uma instância [12].

Essa API também permite inspecionar e manipular metainformações, como as *annotations* [12].

Annotations, ou Anotações, são metadados que podem ser acoplados a vários elementos de codificação para posterior recuperação. Elas incorporam informações adicionais ao código, chamadas de metainformações e diminuem a necessidade de arquivos de configuração externos.

Mesmo não sendo muito utilizadas no cotidiano do desenvolvimento, as anotações são relativamente simples de serem criadas, similares às declarações de interfaces convencionais, bastando apenas adicionar o símbolo "@" precedido do nome da anotação a ser utilizada. Algumas anotações podem ser inseridas sem nenhuma informação adicional, outras, no entanto, utilizam atributos que servem para incrementar o efeito da anotação no código no momento da execução. Para usar as anotações é preciso definir a forma que elas serão lidas, podendo ser diretamente do código-fonte, arquivos de classes ou em tempo de execução (*runtime*), sendo este último o mais utilizado [13] e que será utilizado para o desenvolvimento do estudo de caso.

V. ESTUDO DE CASO

Foi proposto desenvolver um módulo de um Sistema de Planos de Saúde aplicando os conceitos das tecnologias citadas, desenvolvendo também um Autômato de Estados Finitos Adaptativo para ilustrar o funcionamento desse módulo.

O módulo realiza o cálculo da mensalidade de um plano de saúde a ser adquirido por um beneficiário em potencial, em tempo de execução, de acordo com os parâmetros relacionados ao plano de saúde escolhido.

O cálculo da mensalidade para a aquisição de um plano de saúde pode ser formado por vários parâmetros e efetuado de maneira bem simples. Para este estudo de caso foram escolhidos alguns parâmetros básicos: *tipo de contratação*, *sexo*, *data de nascimento* e *módulos*. O cálculo realizado é bem simples.

Um valor de R$ 80,00 será usado como base e será alterado de acordo com os parâmetros citados acima. Por exemplo, se o tipo de contratação for *Pessoa Jurídica*, esse valor já irá sofrer um desconto de 20%.

Para cada um dos parâmetros, têm-se as seguintes possibilidades de escolha:

- *Tipo de Contratação*: Pessoa Física e Pessoa Jurídica;
- *Sexo*: Feminino e Masculino;
- *Data de Nascimento*: o usuário informará sua data de nascimento, que será enquadrada em uma faixa etária;
- *Módulos*: existe a possibilidade de o usuário escolher de um a três módulos, sendo Consulta, Terapias, Exames;

A ilustração pode ser observada no Autômato de Estados Finitos Adaptativo a seguir:

Tomando a linguagem *L = {[Pessoa Física | Pessoa Jurídica].[Feminino | Masculino].[Faixa 1-Faixa 10].[?Consulta ?Terapias ?Exames]}* e a máquina $M = (\{q_0, q_1, q_2, \dots, q_f\}, \Sigma, q_0, \delta, \{F_1, F_2, F_3\})$, onde $\sum$ = *{Pessoa Física, Pessoa Jurídica, Feminino, Masculino, Faixa 1 ... Faixa 10, Consulta, Terapias, Exames}* e $\delta: \{\delta(q_0, tipoC.F_1()) = q_1 ; \delta(q_0, \varepsilon) = q_0\}$ e

$F_1() = \{?\ q_x, \varepsilon, q_y\ ; -q_x, \varepsilon, q_y\ ;$
$+ *l, \varepsilon, l\ ; ?\ q_w, tipoC.F_1, q_z\ ; +q_z, sexo.F_2, l\},$
$F_2() = \{?\ q_x, \varepsilon, q_y\ ; -q_x, \varepsilon, q_y\ ; +*m, \varepsilon, m\ ;$
$+q_x, faixa.F_3, m\},$
$F_3() = \{?\ q_x, \varepsilon, q_y\ ; -q_x, \varepsilon, q_y\ ;$
$+*n, btnCalcular, *o\ ; +q_x, mod, n\ ; +n, \varepsilon, n\},$
temos o seguinte autômato inicial, ilustrado na Figura 4.

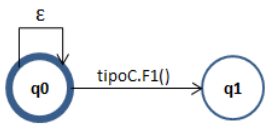

Figura 4: Autômato no estado inicial q_0.

O autômato simula o funcionamento da aplicação, onde começa com apenas dois estados, q_0, que é o estado inicial, onde o usuário ainda não fez uma interação e q_1, que é o estado para onde o sistema vai quando o usuário informa o tipo de contratação. Essa transação entre os estados q_0 e q_1 "*tipoC*" é dotada de uma função adaptativa $F_1()$, assim, ao escolher o tipo de contratação, o autômato vai sofrer uma adaptação por causa de $F_1()$, adquirindo um novo estado e permitindo a escolha de outro parâmetro pelo usuário, no caso, o sexo. Na Figura 5 está ilustrado o autômato no estado em que o tipo de contratação foi escolhido pelo usuário.

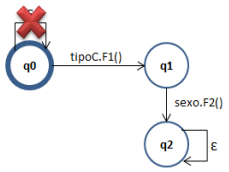

Figura 5: Autômato no estado q_1 da aplicação.

Estando no estado q_1, é possível a escolha do sexo. Essa transação entre os estados q_1 e q_2 "*sexo*" é dotada de uma função adaptativa $F_2()$, assim, ao escolher o sexo, o autômato vai sofrer uma adaptação por causa de $F_2()$, adquirindo um novo estado e permitindo a escolha de outro parâmetro pelo usuário, no caso, a data de nascimento, que será enquadrada em uma faixa etária. Na Figura 6 está ilustrado o autômato no estado em que o sexo foi escolhido pelo usuário.

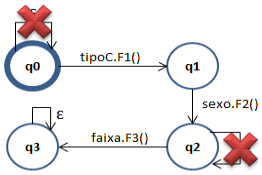

Figura 6: Autômato no estado q_2 da aplicação.

Estando no estado q_2, é possível a escolha da data de nascimento. Essa transação entre os estados q_2 e q_3 "*faixa*" é dotada de uma função adaptativa $F_3()$, assim, ao escolher a data de nascimento, o autômato vai sofrer uma adaptação por causa de $F_3()$, adquirindo um novo estado e permitindo a escolha de outro parâmetro pelo usuário, no caso, o módulo de cobertura e já permite o clique no botão Calcular após a escolha do módulo. Na Figura 7 está ilustrado o autômato no estado em que a data de nascimento foi informada pelo usuário.

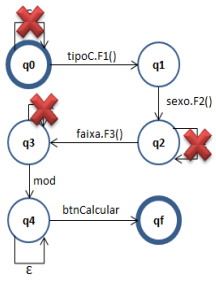

Figura 7: Autômato no estado q_3 da aplicação.

Após a escolha do módulo de cobertura, o autômato estará no estado q_4 e é possível clicar no botão Calcular para obter o valor da mensalidade.

VI. IMPLEMENTAÇÃO

Cada parâmetro escolhido/informado pelo usuário do sistema possui um valor definido nas anotações (*annotations*) e a aplicação montará uma equação em tempo de execução, que resultará no valor da mensalidade do plano de saúde escolhido. Na Figura 8 é possível verificar a definição da anotação *@RegraValor*.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface RegraValor {

    String descricao();

    String valor() default "";

    float fator();

    float vInicial() default 0f;

    float vFinal() default 0f;

}
```

Figura 8: Anotação para definição de valores.

Essa anotação será usada como *metainformação* dos atributos da classe VO, que foi chamada de *CalculoVO*. É interessante verificar que essa anotação é dotada de uma anotação *@Retention(TetentionPolicy.RUNTIME)*, que significa que ela pode ser acessada através da reflexão em tempo de execução (*RUNTIME*). Se não for definida esta diretiva, a anotação não estará disponível através da reflexão.

Na Figura 9, está exemplificado a definição do atributo *dataNascimento*, com sua anotação:

```
@ListaRegras(regras = {
        @RegraValor(desc = "Dt Nascimento", vIni = 1f,  vFin = 18f, fator = 0.4f),
        @RegraValor(desc = "Dt Nascimento", vIni = 19f, vFin = 23f, fator = 1.1f),
        @RegraValor(desc = "Dt Nascimento", vIni = 24f, vFin = 28f, fator = 1.6f),
        @RegraValor(desc = "Dt Nascimento", vIni = 29f, vFin = 33f, fator = 2.0f),
        @RegraValor(desc = "Dt Nascimento", vIni = 34f, vFin = 38f, fator = 2.8f),
        @RegraValor(desc = "Dt Nascimento", vIni = 39f, vFin = 43f, fator = 3.0f),
        @RegraValor(desc = "Dt Nascimento", vIni = 44f, vFin = 48f, fator = 3.4f),
        @RegraValor(desc = "Dt Nascimento", vIni = 49f, vFin = 53f, fator = 3.9f),
        @RegraValor(desc = "Dt Nascimento", vIni = 54f, vFin = 58f, fator = 4.6f),
        @RegraValor(desc = "Dt Nascimento", vIni = 59f, vFin = 999f, fator = 5.6f)
private String dataNascimento;
```

Figura 8: Anotação no atributo dataNascimento.

Esse atributo *dataNascimento* é privado (*private*), por questões de encapsulamento, é uma *String*, e, possui *metainformações* em função da anotação *@ListaRegras*. Não é possível colocar mais de uma anotação do mesmo tipo em um atributo, por isso, na Figura 8 é possível observar que as anotações *@RegraValor* são elementos de um vetor regras, que é definido na anotação *@ListaRegras*. Assim, tem-se *metainformação* de uma *metainformação*. Isso foi necessário devido à regra de negócio que gira em torno desse atributo, que é a definição de valores diferentes para cada faixa etária e foi adotado um número de dez faixas etárias para exemplo.

Os elementos de *@RegraValor*, *descrição*, *valor inicial*, *valor final* e *fator* são usados para que o componente reflexivo realize os cálculos. Como metainformações são informações

das informações, quando o componente reflexivo recuperar o atributo *dataNascimento*, ele terá a possibilidade de recuperar também a anotação para realizar processamento.

Os demais atributos também estão declarados na classe *CalculoVO*. Na Figura 9 está a definição do atributo *tipoContratacao*:

```
@ListaRegras(regras = {
        @RegraValor(descricao = "Tipo de Contratação", valor = "F", fator = 1f),
        @RegraValor(descricao = "Tipo de Contratação", valor = "J", fator = 0.8f) })
private String tipoContratacao;
```

Figura 9: Anotação no atributo tipoContratacao.

Para esse atributo foram usados outros métodos da anotação *@RegraValor*. Não foram usados o *VIni* e o *VFin*, mas sim o *valor*. Então a parte do componente reflexivo responsável pela leitura de atributos com esse tipo de anotação, deve ser diferente da parte responsável pela leitura de um atributo como o *dataNascimento*, por exemplo.

O atributo *modulos* possui uma anotação semelhante a do atributo *dataNascimento*, mas ele é uma lista, portanto, o componente reflexivo deve ter outra parte responsável por ler uma lista.

O componente reflexivo recebeu o nome de *MensalidadeCalculator*, e pode ter uma parte vista na Figura 10:

```
import java.lang.reflect.Field;
import java.util.List;
import vo.CalculoVO;
import annotations.ContentValidator;
import annotations.ListaRegras;
import annotations.RegraValor;

public class MensalidadeCalculator {

    public static float mensalidade(Object o) throws Exception {

        float mensalidade = 80f;

        Class<?> klass = o.getClass();
```

Figura 10: Trecho da classe MensalidadeCalculator.

Como essa classe é a responsável por realizar a reflexão computacional, ela deve usar o pacote *java.lang.reflect*, e, o *.Field* serve para ela refletir os atributos. Ela recebe um parâmetro "*o*" do tipo *Object*, o que a torna adaptativa, pois, não importa se na chamada dela for passado um beneficiário, um aluno, um carro ou qualquer outra coisa, ela vai entender como um *Object* e vai realizar reflexão da mesma maneira, pois através do "*o.getClass()*" o componente realiza reflexão analisando o próprio código do programa em tempo de execução e recupera a classe do objeto.

A declaração da variável "*float mensalidade = 80f;*" vai servir como mensalidade básica, é ela que vai sofrer os cálculos ao longo do tempo de execução, na medida em que o componente reflexivo percorrer os atributos e suas anotações.

A partir do momento que o componente reflexivo recupera a classe do objeto, é possível também recuperar as anotações e os atributos definidos nessa classe, conforme Figura 11:

```
ListaRegras fieldAnnotation = field
        .getAnnotation(ListaRegras.class);

if (fieldAnnotation != null) {
    RegraValor[] regras = fieldAnnotation.regras();

    if (regras != null && regras.length > 0) {
        Object oValor = field.get(o);
```

Figura 11: Recuperação das anotações e atributos.

Esse trecho de código do componente reflexivo é responsável por recuperar a anotação *@ListaRegras*, verificar se existe essa anotação e então definir um vetor regras para obter as anotações *@RegraValor* e caso esse vetor for diferente de nulo e seu tamanho for maior que zero, o componente recupera os campos. Não importa se a classe do objeto possuir um ou mil atributos, o componente reflexivo vai ler todos automaticamente.

A partir dessa etapa, já é possível começar a trabalhar com os atributos recuperados em tempo de execução. Como dito anteriormente, é preciso criar partes diferentes no componente reflexivo para tratar os tipos de atributos que poderão ser obtidos. Uma parte da análise pode ser vista na Figura 12, onde o software verifica se o atributo é do tipo lista.

```
if (field.getType().equals(List.class)) {
    List<?> objects = (List<?>) oValor;
```

Figura 12: Verificação do tipo de atributo lido.

Após esse teste, o componente já realiza a verificação das metainformações obtidas através das anotações desse atributo, imprime no console da *IDE* de desenvolvimento Eclipse uma mensagem "*Regra encontrada:* ", que contém a descrição do atributo e seu fator para o cálculo, com fins de informação. Em seguida, ele atualiza o cálculo da mensalidade, conforme Figura 13:

```
for (RegraValor regraValor : regras) {
    if (regraValor.vInicial() != 0f
            && regraValor.vFinal() != 0f) {
        if (valorCalculo >= regraValor
                .vInicial()
                && valorCalculo <= regraValor
                        .vFinal()) {
            System.out
                    .println("Regra encontrada: "
                            + regraValor
                                    .descricao()
                            + " = "
                            + regraValor
                                    .fator());

            mensalidade += regraValor
                    .fator();
```

Figura 13: Atualização do cálculo da mensalidade.

Isso é feito para os todos os tipos de atributos citados. Esse método de programação, a programação reflexiva, permite ao software fazer uso da tecnologia adaptativa, pois, a classe *CalculoVO* pode sofrer inúmeras alterações por questões de regras de negócio e a classe *MensalidadeCalculator*, que é o componente reflexivo, não sofrerá mudança de uma linha sequer. Isso é muito importante para o programador, pois simplifica muito a manutenção do código do programa, otimizando seu trabalho.

Simulando uma alteração no software, é possível citar uma alteração nas faixas etárias do plano, por exemplo, de dez faixas para duas faixas. Para isso, basta realizar alteração apenas na anotação do atributo *dataNascimento* (diminuindo a quantidade de *@RegraValor* de dez para dois sobre o atributo), que o componente reflexivo vai se adaptar automaticamente. Isso vale também para um caso de alterações dos parâmetros, como por exemplo, a exclusão do atributo sexo, ou a criação de um atributo como *formadorOpiniao*, nenhuma linha do componente reflexivo precisará ser alterada, pois ele analisa a classe e recupera os atributos em tempo de execução, assim, a quantidade de atributos é irrelevante.

Cada campo é liberado apenas quando o anterior é informado, assim, o campo Sexo só será habilitado quando o campo Tipo de Contratação for informado e assim sucessivamente, como ilustrado no Autômato de Estados Finitos Adaptativo no Capítulo V.

Quando todos os parâmetros forem alimentados, o botão *Calcular* será liberado e será possível clicar nele para obter o valor da mensalidade que foi efetuado de acordo com as anotações definidas na classe *CalculoVO*.

VII. CONCLUSÃO

Durante o desenvolvimento deste trabalho foram apresentados conceitos e ferramentas que possibilitam implementar sistemas reflexivos, apresentando características, vantagens e desvantagens do modelo reflexivo de programação. A utilização da programação reflexiva permite a otimização do código fonte e a clareza das regras de negócio, podendo ser utilizada em diversos campos do desenvolvimento de softwares.

Foi desenvolvido um modelo de Autômato de Estados Finitos Adaptativos, que é muito utilizado em Teoria da Computação como modelo matemático que descreve máquinas automáticas, e, também foi desenvolvido um estudo de caso onde os conceitos da Tecnologia Adaptativa e Reflexão Computacional puderam ser bem aplicados. O módulo de um sistema de planos de saúde implementado ficou simples e objetivo, e, a parte teórica, tanto das tecnologias utilizadas quanto do Autômato de Estados Finitos Adaptativo, poderá servir como fonte de estudos para alunos de graduação e demais pesquisadores com o mesmo interesse.

REFERÊNCIAS

[1] PISTORI, Hemerson. Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações. 191p. Tese (Doutorado) - Universidade de São Paulo, São Paulo, 2003.

[2] PAVAN, Willingthon. Tolerância a Falhas e Reflexão Computacional num Ambiente Distribuído. 86p. Dissertação (Mestrado) – Instituto de Informática - Universidade Federação do Rio Grande do Sul, Rio Grande do Sul, Porto Alegre, 2000.

[3] NETO, João José. Um Levantamento da Pesquisa em Técnicas Adaptativas na EPUSP. 2011. 25p. Revista de Sistemas e Computação, Salvador, v. 1, n. 1, p. 23-47, jan./jun. 2011.

[4] NETO, João José Um Levantamento da Evolução da Adaptatividade e da Tecnologia Adaptativa. Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. (p. 496-505).

[5] ROCHA, R. L. A. e Neto, J. J. Construção e Simulação de Modelos Baseados em Autômatos Adaptativos em Linguagem Funcional. Proceedings of ICIE 2001 - International Congress on Informatics Engineering, Buenos Aires: Computer Science Department - University of Buenos Aires, p. 509-521, 2001.

[6] BARTH, Fabrício Jailson. Utilização da Reflexão Computacional Para Implementação de Aspectos Não Funcionais Em Um Gerenciador de Arquivos Distribuídos. 2000. 90p. Monografia (Bacharelado em Ciência da Computação) - Universidade Regional de Blumenau.

[7] BRITO, Elcio Rodrigues. Desenvolvimento de Aplicações Comerciais em Java Usando Reflection. 2012. 36p. Monografia (Bacharelado em Ciência da Computação) – Fundação Educacional do Município de Assis– FEMA/Instituto Municipal de Ensino Superior de Assis - IMESA.

[8] SOUSA, Fabio Cordova de. Utilização da Reflexão Computacional Para Implementação de Um Monitor de Software Orientado a Objetos em Java. 2002. 53p. Monografia (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais – Universidade Regional de Blumenau.

[9] SENRA, Rodrigo Dias Arruda. Programação Reflexiva Sobre o Protocolo de Meta-Objetos Guaraná. 164p. Dissertação (Mestrado) – Instituto de Computação – Universidade Estadual de Campinas, Campinas, 2003.

[10] CORREA, Sand Luz. Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos. 116p. Dissertação (Mestrado) – Instituto de Computação – Universidade Estadual de Campinas, Campinas, 1997.

[11] SOLIS, Daniel M. Illustrated C# 2010.Apress, 2010.

[12] FARTO, Guilherme de Cleva. Introdução à metaprogramação com Java Reflection API. Universidade Federal de São Carlos. Disponível em <http://www.slideshare.net/guilherme_farto/introduo-metaprogramao-com-java-reflection-api>. Acesso em: 31 Set. 2013.

[13] ZAGO, Adams Willians Alencr. Criando Anotações em Java. Universidade Presidente Antônio Carlos. Disponível em < http://www.devmedia.com.br/criando-anotacoes-em-java/22054>. Acesso em: 02 Out 2013.

Diego Augusto Passareli nasceu na cidade de Cândido Mota, São Paulo, Brasil, em 14 de Agosto de 1987. Possui graduação em Licenciatura Plena em Matemática pela Fundação Educacional do Município de Assis (2008) e é estudante do último ano de Bacharelado em Ciência da Computação na Fundação Educacional do Município de Assis (FEMA) em Assis, São Paulo, entre os anos de 2010 e 2013. Suas principais áreas de pesquisas são: Aplicação do Excel para modelagem de indicadores, Autômatos, Algoritmos e desenvolvimento de Aplicações Java.

Almir Rogério Camolesi possui graduação em Processamento de Dados pela Fundação Educacional do Município de Assis (1992), mestrado em Ciência da Computação pela Universidade Federal de São Carlos (2000) e doutorado em Engenharia de Computação e Sistemas Digitais pela Universidade de São Paulo (2007). Atualmente é professor titular da Fundação Educacional do Município de Assis. Tem experiência na área de Ciência da Computação, com ênfase em Tecnologias Adaptativas, atuando principalmente nos seguintes temas: Tecnologia Adaptativa, Computação Distribuída, Teoria da Computação, Modelagem Abstrata, Ensino a Distância, Algoritmos e Programação para Web.

Diomara Martins Reigato Barros possui Especialização em Sistemas de Informação pela Universidade Federal de São Carlos (1996) e graduação em Tecnologia em Processamento de Dados pela Fundação Educacional do Município de Assis - FEMA (1994). Atualmente é Analista de Sistemas e Professora da Fundação Educacional do Município de Assis (FEMA), no curso de Ciência da Computação, nas disciplinas de Teoria da Computação e Compiladores, no curso de Administração, na disciplina de Sistemas de Informação e no curso Análise e Desenvolvimento de Sistemas, na disciplina de Introdução a Computação. Possui experiência na área de Ciência da Computação, com ênfase em Sistemas de Computação atuando nos seguintes temas: Teoria da Computação, Compiladores e Tecnologia Adaptativa.