



Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

DIEGO AUGUSTO PASSARELI

**APLICAÇÃO DA TECNOLOGIA ADAPTATIVA E REFLEXÃO
COMPUTACIONAL EM UM MÓDULO DE SISTEMA DE PLANOS DE
SAÚDE**

ASSIS – SP

2013

DIEGO AUGUSTO PASSARELI

**APLICAÇÃO DA TECNOLOGIA ADAPTATIVA E REFLEXÃO
COMPUTACIONAL EM UM MÓDULO DE SISTEMA DE PLANOS DE
SAÚDE**

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.

Orientadora: Esp. Diomara Martins Reigato Barros

Área de Concentração: Tecnologia Adaptativa, Reflexão Computacional

ASSIS – SP

2013

DEDICATÓRIA

Dedico este trabalho aos meus pais Alice Rodrigues Passareli e Édio Passareli, por terem me tornado o homem que sou. À minha esposa Aline de Lima Trettel Passareli, por toda compreensão, auxílio e amor.

*"A mesma brisa passa
Pelos pinheiros da montanha
E pelos carvalhos do vale;
Então, por que produzem notas
diferentes?" (Um monge taoísta).*

AGRADECIMENTOS

Agradeço a Deus e à minha família, por sempre terem me proporcionado tudo o que precisei para chegar até aqui.

À minha orientadora Diomara Martins Reigato Barros, ao meu coorientador Almir Rogério Camolesi e ao professor Guilherme de Cleve Farto, por todo apoio e tempo investido à minha orientação.

Aos demais professores, por todo ensinamento transmitido ao longo da minha formação.

Aos meus amigos e a todos que direta ou indiretamente contribuíram para a realização deste trabalho.

RESUMO

Este trabalho apresenta conceitos de Tecnologia Adaptativa e sua aplicação no mundo dos softwares por meio da Reflexão Computacional. Aplicar o conceito de adaptatividade para resolução eficiente de certos problemas é a referência da Tecnologia Adaptativa, pois ela possibilita a adaptação, de forma autônoma, de um sistema ou dispositivo adaptativo ao detectar situações que exijam mudanças nas reações em resposta aos estímulos de entrada. Qualquer ação executada por um sistema computacional sobre si próprio pode ser definida como reflexão, que é a capacidade de um programa de reconhecer detalhes internos em tempo de execução que não estavam disponíveis no momento da compilação. Com esses conceitos como base, o desenvolvimento de um estudo de caso foi realizado utilizando programação reflexiva na linguagem Java, estudo de caso que 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.

Palavras-chave: Tecnologia Adaptativa; Reflexão Computacional; Programação Reflexiva.

ABSTRACT

This paper presents concepts of adaptive technology and its application in the world of software through Computational Reflection. Applying the concept of adaptivity for efficient resolution of certain problems is the reference of the adaptive technology, since it enables the adaptation autonomously an adaptive device or system to detect situations that require changes in reactions in response to input stimulus. Any action performed by a computer system on itself can be defined as reflection, which is the ability of a program to recognize internal details at runtime that were not available at compile time. With these concepts as a basis, the development of a case study was conducted using reflective programming in the Java language, case study comes from one screen to calculate the monthly payment of a health plan, where the advantages to the developer are very large, such as ease in maintaining the source code and optimization work.

Keywords: Adaptive Technology; Computational Reflection; Reflective Programming.

LISTA DE ILUSTRAÇÕES

Figura 1 - Estrutura geral de um dispositivo Adaptativo	17
Figura 2 - Autômato inicial dotado de uma função adaptativa	19
Figura 3 - Autômato se adaptando de acordo com os estímulos de entrada	20
Figura 4 - Visualização genérica de um sistema computacional reflexivo	22
Figura 5 - Outro exemplo de reflexão computacional	23
Figura 6 - Torre reflexiva	25
Figura 7 - Modelo de metaclasses	26
Figura 8 - Modelo de metaobjetos	27
Figura 9 - Modelo de metacomunicações	27
Figura 10 - Estrutura de funcionamento Java	29
Figura 11 - Interface da aplicação em seu estado inicial	40
Figura 12 - Autômato no estado inicial q_0 da aplicação	41
Figura 13: Interface da aplicação com o autômato no estado q_1 da aplicação	42
Figura 14: Interface da aplicação com o autômato no estado q_2 da aplicação	42
Figura 15: Interface da aplicação com o autômato no estado q_3 da aplicação	43
Figura 16: Plano escolhido e mensalidade calculada	44

LISTA DE CÓDIGOS

Código 1 – Anotação para definição de valores	35
Código 2 – Anotação no atributo dataNascimento	36
Código 3 – Anotação no atributo tipoContratacao	37
Código 4 – Trecho da classe MensalidadeCalculator	37
Código 5 – Recuperação das anotações e atributos	38
Código 6 – Verificação do tipo de atributo lido	39
Código 7 – Atualização do cálculo da mensalidade	39

LISTA DE ABREVIATURAS E SIGLAS

API: *Application Programming Interface*

JDK: *Java Development Kit*

JEE: *Java Enterprise Edition*

JME: *Java Micro Edition*

JSE: *Java Standard Edition*

JVM: *Java Virtual Machine*

SO: *Sistema Operacional*

TA: *Tecnologia Adaptativa*

VO: *Value Object*

SUMÁRIO

1. INTRODUÇÃO	12
1.1 OBJETIVOS.....	12
1.2 JUSTIFICATIVAS	13
1.3 MOTIVAÇÃO	13
1.4 PERSPECTIVAS DE CONTRIBUIÇÃO.....	14
1.5 METODOLOGIAS DE PESQUISA	14
2. TECNOLOGIA ADAPTATIVA.....	15
2.1 DISPOSITIVOS ADAPTATIVOS	16
3. REFLEXÃO COMPUTACIONAL	21
3.1 ARQUITETURA REFLEXIVA	24
3.2 MODELOS DE REFLEXÃO.....	26
4. TECNOLOGIA JAVA	28
4.1 JVM.....	29
4.2 REFLECTION	30
4.3 ANNOTATION	31
5. PROPOSTA DO TRABALHO	34
5.1 APLICAÇÃO DA PROPOSTA	34
6. CONSIDERAÇÕES FINAIS	45
6.1 RESULTADOS ALCANÇADOS.....	45
6.2 TRABALHOS FUTUROS.....	46
REFERÊNCIAS.....	47

1. INTRODUÇÃO

É comum encontrar no meio comercial, sistemas que não se modificam, em comportamento e estrutura, para solucionar um problema ou em situações inesperadas. Independente de seus estímulos de entrada, seu comportamento é sempre o mesmo - fazer somente o que lhes foi programado.

A Tecnologia Adaptativa (TA) 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 (PISTORI, 2003).

Mas há dificuldades em realizar a codificação da TA devido às características não adaptativas da maioria das linguagens de programação. No entanto, existem algumas técnicas de programação, como a promissora Reflexão Computacional (que é obtida quando o software interrompe sua execução para fazer análises/computações sobre si mesmo, podendo então decidir se terá ou não que mudar sua execução (PAVAN, 2000)), que podem implementar a TA.

1.1 OBJETIVOS

O objetivo desse trabalho é realizar um estudo sobre os conceitos de Tecnologia Adaptativa, buscando desenvolver um estudo de caso empregando esta tecnologia. Para o estudo de caso, foi desenvolvido um software com características adaptativas com foco na gestão de planos de saúde, utilizando no desenvolvimento desse software técnicas de programação reflexiva.

1.2 JUSTIFICATIVAS

Os conceitos de Reflexão Computacional e Tecnologia Adaptativa não são amplamente estudados e utilizados, devido às limitações das linguagens de programação e ao fato de serem assuntos relativamente novos.

A Tecnologia Adaptativa possibilita a um dispositivo adaptativo se modificar ao detectar situações que exijam mudanças nas reações em resposta aos estímulos de entrada.

O emprego da Tecnologia Adaptativa em sistemas comerciais, que em sua maioria apresentam estruturas fixas, é possível graças a técnicas de programação que os 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 (PAVAN, 2000).

O intuito desse trabalho, então, é apresentar os conceitos de Tecnologia Adaptativa, Reflexão Computacional e suas aplicabilidades, desenvolvendo um estudo de caso utilizando programação reflexiva em Java.

1.3 MOTIVAÇÃO

O estudo sobre Tecnologia Adaptativa e Reflexão Computacional abre um grande leque de possibilidades para outros assuntos relacionados, como na área de inteligência artificial (JÚNIOR, 2012), de robótica, visão computacional e reconhecimento de padrões a partir de representações sintáticas de entes geométricos (NETO et al., 2000 apud PISTORI, 2003, p. 2), contribuindo para a área da pesquisa, que é pouco explorada atualmente pelos alunos de graduação.

1.4 PERSPECTIVAS DE CONTRIBUIÇÃO

Este trabalho visa ampliar a bibliografia sobre assuntos relacionados à Tecnologia Adaptativa e Reflexão Computacional, com um pouco de teoria e um exemplo de aplicação em Java. Ao final deste trabalho espera-se que as contribuições produzidas possam ser transformadas em um artigo científico para um evento relacionado nas áreas de foco desta pesquisa.

1.5 METODOLOGIAS DE PESQUISA

Este trabalho foi realizado com a metodologia experimental. Primeiramente foi feito um estudo de livros, artigos, teses e Internet buscando adquirir e ampliar conhecimento sobre Tecnologia Adaptativa, Reflexão Computacional e recursos da programação reflexiva em Java. Após esta etapa foi implementado o estudo de caso proposto com as tecnologias estudadas.

2. TECNOLOGIA ADAPTATIVA

Entende-se por adaptativo o que realiza adaptação, ou melhor, que possui a capacidade de se adaptar, em função de estímulos, para resolver alguma situação.

A programação de sistemas com capacidade de realizar automodificações surgiu com os primeiros computadores, com o intuito de economizar a memória principal, então escassa e cara. Todavia, essa prática era pouco compreendida pelos programadores e a manutenção desses programas se tornava difícil, acarretando baixa confiabilidade (NETO, 2011).

Esse conceito acabou então tendo um começo lento e foi evoluindo aos poucos, chegando hoje a ser utilizado em considerações especiais como a Inteligência Artificial.

Aplicar o conceito de adaptatividade para resolução eficiente de certos problemas é a referência da Tecnologia Adaptativa, pois ela possibilita a adaptação, de forma autônoma, de um sistema ou dispositivo adaptativo ao detectar situações que exijam mudanças nas reações em resposta aos estímulos de entrada (TCHEMRA, 2007), com o uso de ações adaptativas, que permitem modificar o conjunto de regras, removendo regras existentes e incluindo novas regras dinamicamente, sem a interferência de qualquer agente externo (NETO, 2007).

É interessante ressaltar que duas instâncias idênticas de um mesmo sistema adaptativo, por exemplo, podem atingir estados finais diferentes de acordo com a diversidade dos estímulos a que forem submetidas em suas operações (NETO, 2007).

Essas ações adaptativas não acrescentam poder adicional à computação, mas melhoram o poder de expressão de atividades computacionais intrincadas.

Algumas áreas de aplicações da Tecnologia Adaptativa são (PISTORI, 2003):

- Robótica – utilizados autômatos adaptativos em automação, planejamento de rotas para trânsito urbano, etc.

- Segurança – utilizados autômatos adaptativos em criptografia e mecanismos de controle de acesso.
- Arte – exploração da composição musical automática, busca de padrões musicais, etc.
- Otimização – otimização de parâmetros, em função da particularidade de cada caso, otimização de rotas em tempo real, etc.
- Tomada de decisão – é um alvo interessante para o emprego da adaptatividade, pois possui um grande leque de alternativas.

2.1 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. (PISTORI, 2003).

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. (PISTORI, 2003).

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. (PISTORI, 2003).

A camada adaptativa é exclusivamente dedicada à descrição dos fenômenos de automodificação que devem ocorrer no dispositivo adaptativo que se deseja construir (NETO, 2011).



Figura 1: Estrutura geral de um dispositivo Adaptativo.

Segundo Pistori (2003), os dispositivos adaptativos podem ser formalizados como uma dupla $AD = (CS_0, CA)$, onde CS_0 representa a camada subjacente em sua forma original e CA representa a camada adaptativa.

A descrição da camada subjacente é dada por (PISTORI, 2003):

$$CS_0 = (\mathcal{C}_0, \Sigma, \Phi, c_0, c_a, \mathbb{R}_0)$$

Onde:

- $\mathcal{C}_0 \subseteq \mathcal{C}$ é o conjunto das possíveis configurações da camada subjacente em sua situação inicial. O conjunto \mathcal{C} possui todas as configurações possíveis para o AD .
- Σ é o conjunto fixo e finito contendo todos os possíveis eventos válidos como estímulo de entrada para AD , incluindo o valor nulo.
- Φ é o conjunto fixo e finito dos possíveis símbolos de saída, incluindo o valor nulo.
- $c_0 \in \mathcal{C}_0$ representa a configuração inicial do dispositivo.
- $c_a \subseteq \mathcal{C}$ é o conjunto de configurações de aceitação.
- $\mathbb{R}_0 \subseteq \mathbb{R} \subseteq \mathcal{C} \times \Sigma \times \mathcal{C} \times \Phi$ é o conjunto de regras da camada subjacente em situação inicial. O conjunto \mathbb{R} , de maneira análoga à \mathcal{C} , contém todas as possíveis regras de um AD , contém as regras iniciais e as que poderão vir a ser inseridas durante uma ação adaptativa. Cada regra $r = (c_i, s, c_j, z) \in \mathbb{R}$ deve ser interpretada da seguinte forma: dado um estímulo de entrada $s \in \Sigma$

e estando na configuração $c_i \in \mathcal{C}$, passa para a configuração $c_j \in \mathcal{C}$, consumindo s e produzindo $z \in \Phi$.

E a descrição da camada adaptativa é dada por (PISTORI, 2003):

$$CA = (\mathcal{R}, \mathcal{A})$$

Onde:

- \mathcal{R} é o conjunto de ações adaptativas, que contém também o valor nulo.
- $\mathcal{A}: \mathbb{R} \rightarrow \mathcal{R}^2$ é uma função que mapeia cada possível regra da camada subjacente em um par ordenado de ações adaptativas. As duas ações adaptativas desse par ordenado devem ser acionadas, respectivamente, antes e depois da execução da regra à qual elas estão associadas. Por isso elas são denominadas ações anterior e posterior.

A execução das ações adaptativas induz uma sequência, $CS_0, CS_1, CS_2, \dots, CS_n$, de transformações da camada subjacente, na qual cada elemento $CS_i, 0 \leq i \leq n$ é definido de maneira análoga à CS_0 . Os elementos \mathcal{C}_i e \mathbb{R}_i , de CS_i , correspondem aos conjuntos de configurações e regras produzidas pela aplicação de uma ação adaptativa $a_i (i > 0)$, sobre CS_{i-1} .

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 (NETO, 2007).

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 (ROCHA, 2001).

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 (PISTORI, 2003).

Um exemplo de Autômato de Estados Finitos Adaptativo é possível ser observado abaixo:

Tomando a linguagem $L = \{a^n b^n c^n | n \geq 0\}$ e a máquina $M = (\{q_0, q_1, q_2, \dots, q_f\}, \{a, b, c\}, q_0, \delta, \{F\})$, onde $\delta: \{\delta(q_0, a, F()) = q_0; \delta(q_0, \varepsilon) = q_f\}$ e $F() = \{? q_x, \varepsilon, q_y; - q_x, \varepsilon, q_y; + q_x, b, * k; + * l, c, q_y; + k, \varepsilon, l\}$, temos o seguinte autômato inicial, ilustrado na Figura 2.

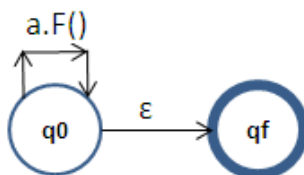
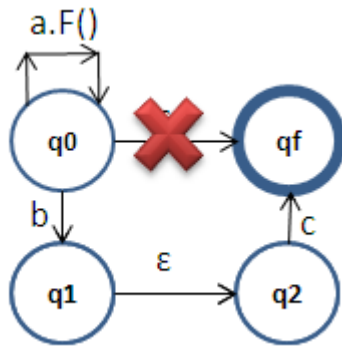


Figura 2: Autômato inicial dotado de uma função adaptativa.

Tomando uma cadeia de caracteres (palavra) $w = aaabbbccc$, o autômato vai se adaptando, de acordo com as leituras efetuadas, baseado na função adaptativa $F()$, conforme ilustrado na Figura 3.

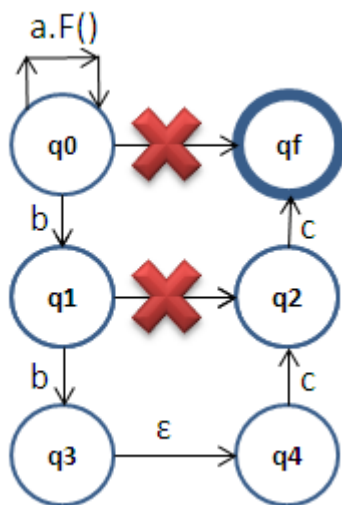
No passo 1 da Figura 3, o autômato lê o primeiro caractere a e realiza a função adaptativa, onde procura a transição que contém o vazio (ε), a remove e cria dois novos estados com transições entre eles, de maneira que a transição com o ε ligue os dois novos estados. Nos passos 2 e 3, como os dois próximos caracteres lidos são a novamente, o autômato realiza o mesmo processo do passo 1.

No quadro do passo 3 da Figura 3 é possível observar que o autômato modificou sua própria topologia, adequando-se à palavra lida e passando de dois para oito estados, atingindo o estado final q_f e aceitando a palavra.



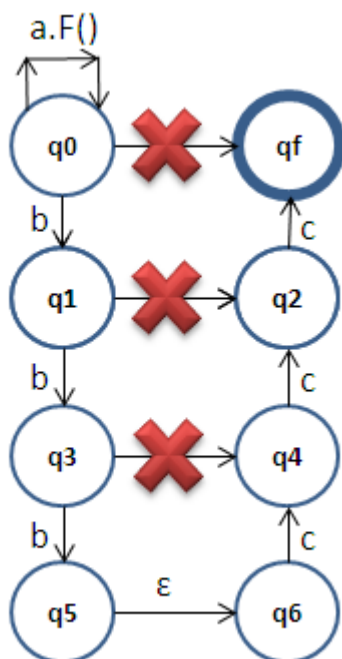
Passo 1

Leitura do primeiro caractere **aaabbbccc** e permanecendo no estado q_0 .



Passo 2

Leitura do segundo caractere **aaabbbccc** e permanecendo no estado q_0 .



Passo 3

Leitura do terceiro caractere **aaabbbccc** e permanecendo no estado q_0 .

O próximo caractere a ser lido é b , assim, o autômato passa para o estado q_1 e não realiza mais a adaptação.

Figura 3: Autômato se adaptando de acordo com os estímulos de entrada.

3. 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 (BARTH, 2000).

A reflexão pode ser definida como qualquer ação executada por um sistema computacional sobre si próprio (BRITO, 2012), em outras palavras, é a capacidade de um programa reconhecer detalhes internos em tempo de execução que não estavam disponíveis no momento da compilação (MAES, 1987 apud BRITO, 2012, p. 15).

Quando um programa reflexivo entra em execução, ele considera variáveis, assim como 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 (SOUSA, 2002).

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 (BRITO, 2012). A representação das informações manipuláveis de um sistema sobre si próprio é chamada de *metainformação* (SENRA, 2003).

Não existe dissociação entre a introspecção e o redirecionamento, mas sim, uma relação de causa e efeito (SENRA, 2003).

Para que o mecanismo de reflexão seja implementado de forma flexível é necessário definir uma *arquitetura reflexiva* (CORREA, 1997). 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* (SOUSA, 2002).

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 (BARTH, 2000).

A Figura 4 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 (BARTH, 2000).



Figura 4: Visualização genérica de um sistema computacional reflexivo (BARTH, 2000).

Exemplificando, em uma aplicação de tempo real responsável por conduzir um objeto por uma esteira por determinado tempo, o nível funcional seria o responsável pela lógica necessária para conduzir o objeto pela esteira, e o nível não funcional seria o responsável pelos aspectos relativos a restrições temporais e de sincronização e exceções temporais, por exemplo (BARTH, 2000).

Na Figura 5, outro exemplo de reflexão computacional é ilustrado, onde um cliente faz uma requisição a um servidor. A reflexão está na estrutura servidor, que é formada por um *componente base* e um *componente de reflexão*, onde esse último, apesar de possuir o mesmo comportamento que o componente base, age interceptando as informações que deveriam ir diretamente ao componente base, podendo alterar as características do componente base antes de repassá-las (BARTH, 2000).

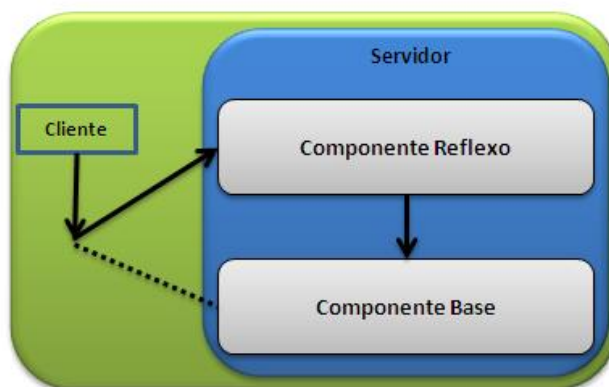


Figura 5: Outro exemplo de reflexão computacional (BARTH, 2000).

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 sanadas 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 (SILVA, 1997).

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 (BARTH, 2000).

3.1 ARQUITETURA REFLEXIVA

A Reflexão Computacional define conceitualmente uma arquitetura em níveis, denominada *arquitetura reflexiva* (BRITO, 2012), que se constitui basicamente de dois níveis: um representando os objetos, o chamado *nível base*, que se refere ao domínio da aplicação e é ideal para implementar requisitos funcionais, e o outro nível se refere a parte reflexiva (autodomínio), chamado *metanível* (CORREA, 1997).

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 (CORREA, 1997; BRITO, 2012).

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} (BARTH, 2000). Essa torre de reflexão pode ser analisada na Figura 6.

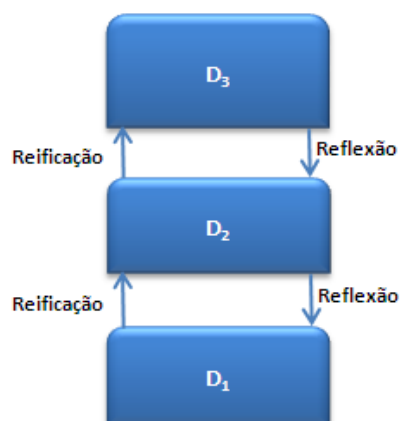


Figura 6: Torre reflexiva (BARTH, 2000).

Na Figura 6, 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 (BARTH, 2000). É 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 (SOUSA, 2002).

3.2 MODELOS DE REFLEXÃO

Segundo Buzato (1998 apud BARTH, 2000, p. 23), os modelos de reflexão computacional podem ser divididos em modelo de *metaclasses*, modelo de *metaobjetos* e modelo de *metacomunicações*.

No modelo de metaclasses, um metaobjeto é compartilhado por todos os objetos que são instâncias da mesma classe (SILVA, 1997), ou seja, todo o objeto instanciado pela mesma classe terá o mesmo comportamento reflexivo de acordo com o que está modelado em sua metaclasses (BARTH, 2000). Como consequência, este modelo é menos flexível que os demais (CORREA, 1997).

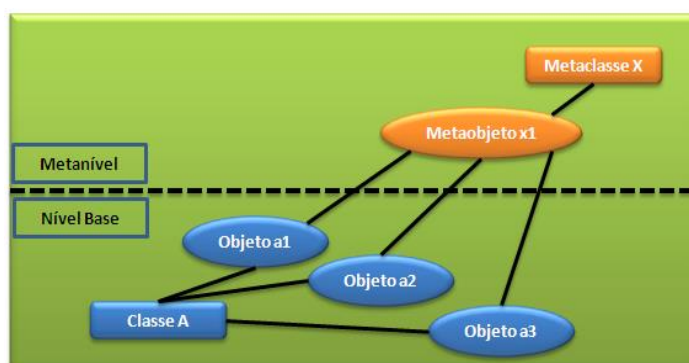


Figura 7: Modelo de metaclasses (BARTH, 2000).

O modelo de metaobjetos se diferencia do modelo de metaclasses principalmente por sua associação por objetos e não por classes de objetos (BARTH, 2000), onde cada objeto pode ser controlado por um metaobjeto correspondente. Esta abordagem é mais flexível que a de metaclasses, pois ela especializa o comportamento do metaobjeto de acordo com o objeto individual (CORREA, 1997).

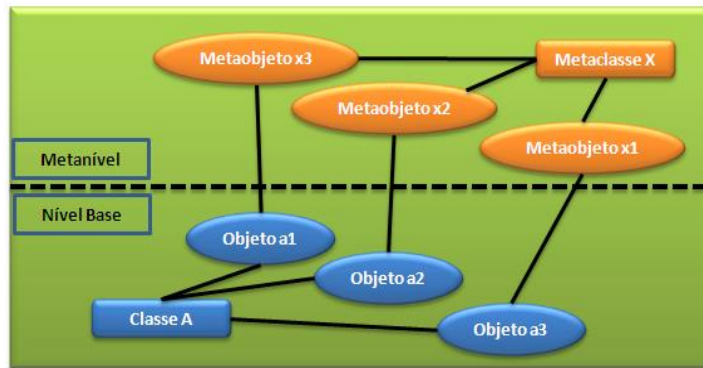


Figura 8: Modelo de metaobjetos (BARTH, 2000).

O modelo de metacomunicação pressupõe a existência de uma classe pré-definida no sistema, por exemplo, *Mensagem* (SILVA, 1997). Quando uma mensagem é enviada, uma instância da classe *Mensagem* é criada, e uma mensagem *send*, que possui todas as informações sobre a mensagem (a que objeto pertence, quais os seus parâmetros, quem está enviando, etc.), é enviada para ela. Este modelo é o mais flexível, mas é também o mais ineficiente, uma vez que o sistema passa a maior parte operando no metanível, já que toda mensagem será interceptada pelo metanível, mesmo que esta mensagem não necessite da adição de um comportamento reflexivo (BARTH, 2000).

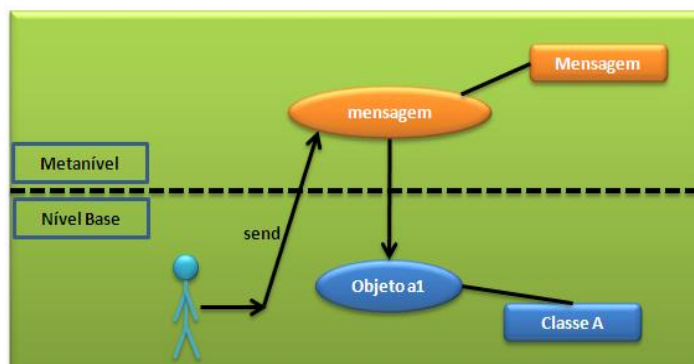


Figura 9: Modelo de metacomunicações (BARTH, 2000).

4. 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 iniciou com um projeto em 1991 sendo baseado nas linguagens de programação C/C++ (DEITEL, 2003).

Essa linguagem de programação possui uma sintaxe amigável e gerenciamento de memória automático, além de portabilidade, o que permite a execução de programas em qualquer SO, por isso Java é muito popular (SIERRA; BATES, 2010).

Java é importante e provavelmente será a base para as futuras gerações de linguagens de computação. A plataforma de computação Java é um conjunto de tecnologias e pode ser dividida em três partes (METSKER, 2004):

- JSE (*Java Standard Edition*): plataforma padrão com desenvolvimento voltado para desktops;
- JEE (*Java Enterprise Edition*): plataforma voltada para desenvolvimento de aplicações de grande porte ou para *Web*;
- JME (*Java Micro Edition*): plataforma voltada para dispositivos móveis ou embarcados.

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, conforme Figura 10.

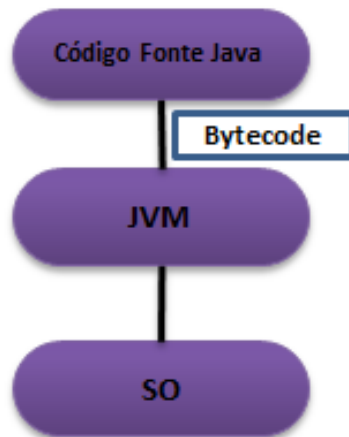


Figura 10: Estrutura de funcionamento Java.

4.1 JVM

A JVM é um programa que carrega e executa aplicativos Java, intermediando a comunicação da aplicação com o hardware. Ela funciona como uma máquina imaginária (*Virtual Machine*), proporcionando portabilidade e segurança, uma vez que qualquer dispositivo munido de uma JVM pode executar qualquer aplicação escrita em Java (LIMA, 2012).

Depois que um programa é compilado pelo JDK (*Java Development Kit* – Contém ambiente de execução, ferramentas para desenvolvimento como compilador, ferramentas de análise de desempenho, etc.) em bytecode, ele pode ser executado pela JVM, pois apresenta uma codificação binária universal disposto em arquivos de acordo com a classe compilada com extensão *.class* (LIMA, 2012).

4.2 REFLECTION

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 (SOLIS, 2010).

Um programa pode olhar para os metadados enquanto ele está em execução e, isso é chamado de *reflection* (reflexão) (SOLIS, 2010). A ênfase da programação reflexiva é a modificação ou a análise dinâmica, podendo modificar sua estrutura e praticar a autoanálise em tempo de execução (JÚNIOR, 2012).

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 (FARTO, 2013).

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*, disponibilizada pela JDK desde a versão 1.1. Essa API também permite inspecionar e manipular metainformações, como as *annotations* (FARTO, 2013).

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 (FARTO, 2013).

A classe *java.lang.Class* representa classes Java de maneira que seja possível obter e manipular informações dinamicamente e a JVM constrói objetos dessa classe quando novas classes são carregadas. Essa classe *java.lang.Class* foi mantida na *java.lang* por motivos de compatibilidade, mas ela poderia pertencer ao pacote *java.lang.reflect* (FARTO, 2013).

Através de uma instância de *java.lang.Class* é possível verificar se o objeto *Class* representa um array, um tipo primitivo como *char*, *byte*, *int* e etc, verificar os modificadores de acesso do objeto *Class*, refletir membros (*Field*, *Method* e *Constructor*) do tipo representado e etc (FARTO, 2013).

4.3 ANNOTATIONS

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. Como exemplo, pode-se citar a necessidade de configurar uma aplicação em tempo de execução (FARTO, 2013).

Mesmo não sendo muito utilizadas no cotidiano do desenvolvimento, as anotações são relativamente simples de serem criadas, similares às declarações de interfaces convencionais, podendo ser aplicadas para anotar declarações, bastando para isso 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 (ZAGO, 2013) e que será utilizado para o desenvolvimento do estudo de caso.

Vários frameworks e até a própria linguagem de programação Java em algumas de suas API's, utilizam anotações para simplificar o código e também para diminuir o acoplamento entre as classes (ZAGO, 2013).

Este tema é útil quando se precisa anotar o código não simplesmente para documentação, mas de maneira que essas marcações possam ser verificadas em tempo de compilação, ou utilizadas por ferramentas, tais como analisadores de código, frameworks de persistência ou de testes unitários, etc. (ARAÚJO, 2013).

As categorias de anotações são (ARAÚJO, 2013):

- **Anotações marcadoras** – aquelas que não possuem membros. São identificadas apenas pelo nome, como por exemplo `@Test`;
- **Anotações de valor único** – são similares às anteriores, no entanto, possuem um único membro, chamado valor. Elas são representadas pelo nome da anotação e um par *nome = valor*, ou simplesmente com o valor, entre parênteses. Ou seja, quando a anotação possui um único membro, só é necessário informar o valor, além do nome da anotação. Por exemplo, `@Test("valor")`;
- **Anotações completas** – são aquelas que possuem múltiplos membros. É preciso usar a sintaxe completa para cada par *nome = valor*. Neste caso, cada par é informado separado do outro por uma vírgula. Por exemplo, `@Test(valor1=1, valor2=0)`.

Em uma classe é possível ter várias anotações, cada uma delas correspondente a algum *tipo anotação*, tal como `@Override`, que se utiliza para marcar os métodos sobrescritos. O tipo anotação é a definição da anotação e a anotação propriamente dita é um caso específico deste tipo. Analogamente, é possível comprar a uma aplicação onde existe uma classe *Pessoa*, que é possível ter várias instâncias dessa

classe ao mesmo tempo. Ou seja, tipo anotação é representado por essa classe e a anotação é como se fosse uma instância da classe (ARAÚJO, 2013).

O desenvolvedor pode definir suas próprias anotações a serem usadas em uma aplicação. Além disso, Java apresenta algumas anotações predefinidas (ARAÚJO, 2013), tais como:

- **@Deprecated**: É utilizada quando algum método tem seu uso desencorajado por ser perigoso ou por ter uma alternativa melhor desenvolvida;
- **@Override**: Usado principalmente em casos de polimorfismo, sempre que for sobrescrever um método da superclasse;
- **@SuppressWarnings ("deprecation")**: Todos os avisos da categoria "inadequado" são ignorados;
- **@SuppressWarnings ("unchecked")**: Todos os avisos da categoria "não verificado" são ignorados;
- **@SuppressWarnings ({"unchecked", "deprecation"})**: Todos os avisos de qualquer categoria são ignorados.

5. PROPOSTA DO TRABALHO

Com o estudo realizado sobre TA, Reflexão Computacional e o método de programação Reflection usado na Tecnologia Java, 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 escolhidos.

5.1 DESENVOLVIMENTO DA PROPOSTA

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 trabalho foram escolhidos alguns parâmetros básicos: *tipo de contratação*, *sexo*, *data de nascimento* e *módulos*. Para cada um desses parâmetros, ou atributos da classe VO (*Value Object*), tem-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*;

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. No Código 1 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;

}
```

Código 1: 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(RetentionPolicy.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.

No Código 2, 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;
```

Código 2: 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, no Código 2 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*. No Código 3 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;
```

Código 3: 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 no Código 4:

```
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();
```

Código 4: 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 Código 5:

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

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

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

Código 5: 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 no Código 6, onde o software verifica se o atributo é do tipo lista.

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

Código 6: 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 Código 7:

```
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();
```

Código 7: 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 *formadorOpiniaio*, 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..

A interface gráfica da aplicação em seu estado inicial pode ser observada na Figura 11, onde o usuário poderá começar informando o tipo de contratação.

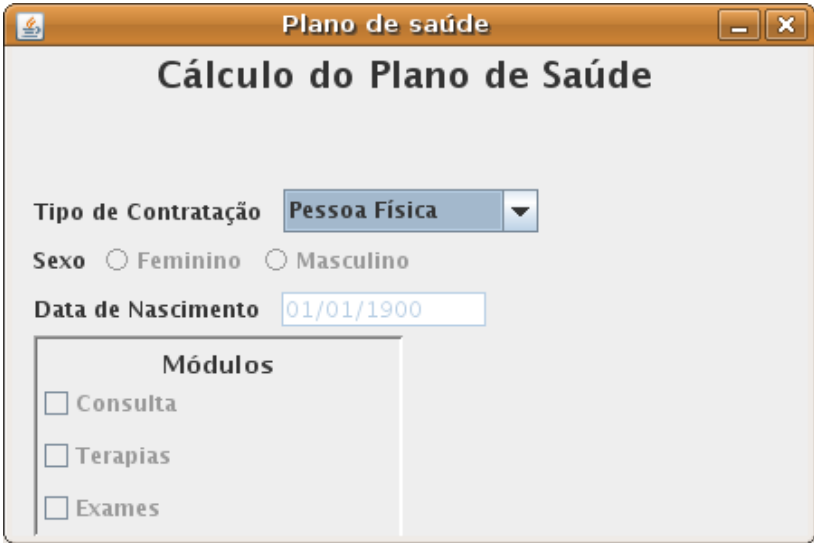


Figura 11: Interface da aplicação em seu estado inicial.

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 a seguir:

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

$$\text{e } \delta: \{\delta(q_0, \text{tipoC}.F_1()) = q_1; \delta(q_0, \varepsilon) = q_0\}$$

$$\text{e } F_1() = \{? q_x, \varepsilon, q_y; - q_x, \varepsilon, q_y; + * l, \varepsilon, l; ? q_w, \text{tipoC}.F_1, q_z; + q_z, \text{sexo}.F_2, l\},$$

$$F_2() = \{? q_x, \varepsilon, q_y; - q_x, \varepsilon, q_y; + * m, \varepsilon, m; + q_x, \text{faixa}.F_3, m\},$$

$$F_3() = \{? q_x, \varepsilon, q_y; - q_x, \varepsilon, q_y; + * n, \text{btnCalcular}, * o; + q_x, \text{mod}, n; + n, \varepsilon, n\},$$

temos o seguinte autômato inicial, ilustrado na Figura 12.

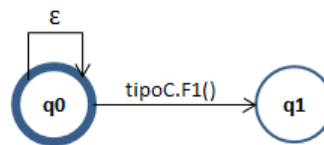


Figura 12: Autômato no estado inicial q_0 da aplicação.

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 transição 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 13 está

ilustrado a interface do sistema e o autômato no estado em que o tipo de contratação foi escolhido pelo usuário.

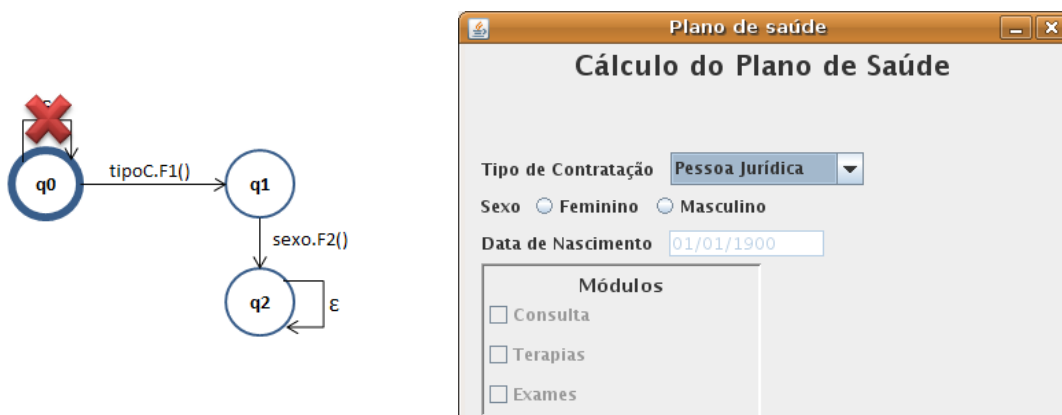


Figura 13: Interface da aplicação com o autômato no estado q_1 da aplicação.

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, definida nas anotações, conforme Código 2. Na Figura 14 está ilustrado a interface do sistema e o autômato no estado em que o sexo foi escolhido pelo usuário.

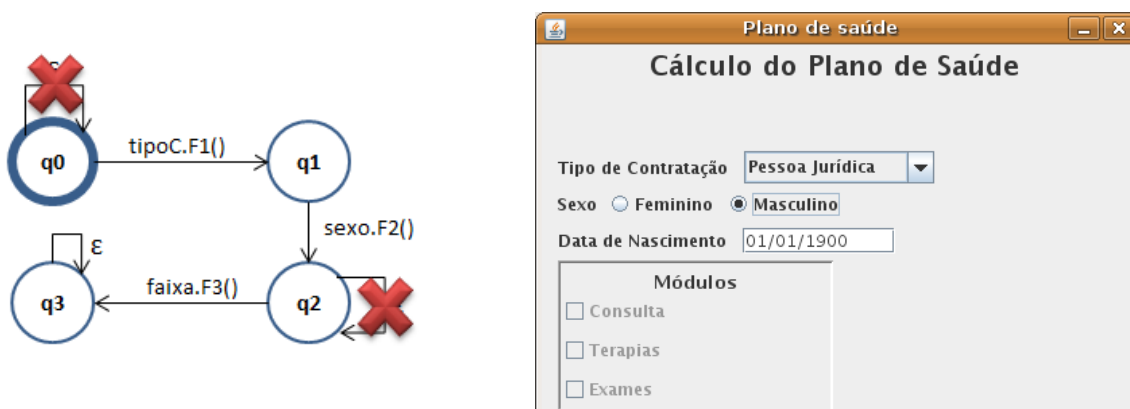


Figura 14: Interface da aplicação com o autômato no estado q_2 da aplicação.

Estando no estado q_2 , é possível a escolha da data de nascimento. Essa transição entre os estados q_2 e q_3 "faixa" é dotada de uma função adaptativa $F_3()$, assim, ao escolher a data de nascimento e por reflexão computacional o programa realizar os cálculos e verificar em qual faixa etária a idade se enquadra, 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 15 está ilustrado a interface do sistema e o autômato no estado em que a data de nascimento foi informada pelo usuário.

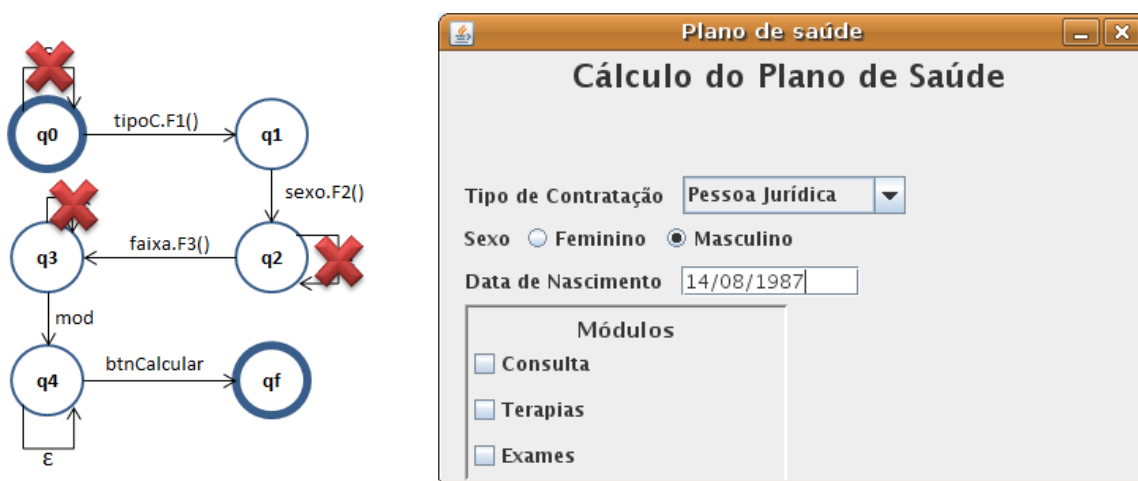
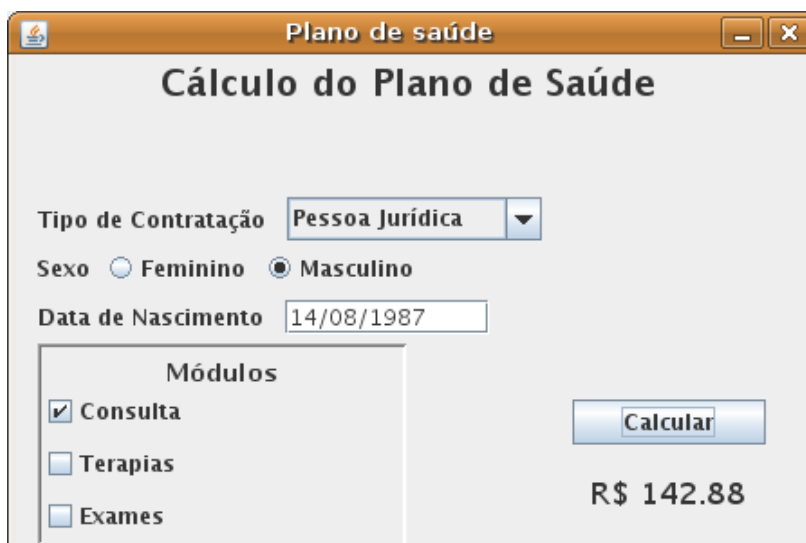


Figura 15: Interface da aplicação com o autômato no estado q_3 da aplicação.

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 que foi efetuado de acordo com as anotações definidas na classe *CalculoVO*. Na Figura 16 está ilustrada a interface do sistema no estado em que todos os parâmetros foram informados pelo usuário e o botão *Calcular* foi acionado.



Plano de saúde

Cálculo do Plano de Saúde

Tipo de Contratação

Sexo Feminino Masculino

Data de Nascimento

Módulos

- Consulta
- Terapias
- Exames

R\$ 142.88

Figura 16: Plano escolhido e mensalidade calculada.

É interessante ressaltar que a interface gráfica da aplicação e a forma de cálculo parecem bem simples e não é possível concluir, em nível de usuário, que existe reflexão computacional e teoria da adaptatividade sendo utilizadas, mas para o desenvolvedor a diferença é notória, graças às vantagens apresentadas no capítulo 4.

6. CONSIDERAÇÕES FINAIS

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.

Foram apresentados também dois modelos de Autômatos de Estados Finitos Adaptativos, que são muito utilizados em Teoria da Computação como modelos matemáticos que descrevem máquinas automáticas.

6.1 RESULTADOS ALCANÇADOS

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 pessoas com o mesmo interesse.

6.2 TRABALHOS FUTUROS

Com base nas tecnologias estudadas e com o aprendizado da programação reflexiva, estima-se desenvolver um artigo científico para um evento relacionado nas áreas de foco desta pesquisa.

REFERÊNCIAS

ARAÚJO, Carlos. **Entendendo Anotações**. Centro Universitário Luterano de Santarém. Disponível em < <http://www.devmedia.com.br/entendendo-anotacoes-revista-easy-java-magazine-25/26772>>. Acesso em: 02 Out. 2013.

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.

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.

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.

DEITEL, H.M. **Java como programar**, trad. Carlos Arthur Lang Lisboa, 4 ed. Bookman, Porto Alegre, 2003.

FARTO, Guilherme de Cleve. **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.

JÚNIOR, José David Camoleze. **Proposta de um Framework para a Persistência de Dados Usando a Tecnologia .NET**. 2012. 26p. Monografia (Bacharelado em Ciência da Computação) – Fundação Educacional do Município de Assis– FEMA/Instituto Municipal de Ensino Superior de Assis - IMESA.

LIMA, Mariana Budiski de. **O Impacto das Boas Práticas de Programação no Gerenciamento de Memória no Java**. 2012. 64p. Monografia (Bacharelado em Ciência da Computação) – Fundação Educacional do Município de Assis– FEMA/Instituto Municipal de Ensino Superior de Assis - IMESA.

METSKER, Steven John. **Padrões de Projeto em Java**. São Paulo: Editora Bookman, 2004. 407 p.

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)

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.

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.

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

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

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.

SIERRA, Kathy e BATES, Bert. **Use a Cabeça! Java**. Rio de Janeiro: Editora Alta Books, 2010. 484 p.

SILVA, Romulo Cesar. **RStabilis: Uma Máquina Reflexiva de Busca**. 111p. Dissertação (Mestrado) – Instituto de Computação – Universidade Estadual de Campinas, Campinas, 1997.

SOLIS, Daniel M. **Illustrated C# 2010**. Apress, 2010.

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.

TCHEMRA, A. H. **Aplicação da Tecnologia Adaptativa em Sistemas de Tomada de Decisão**. Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007 (p. 552-556).

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.