UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO DE CIÊNCIAS AGRÁRIAS

DEPARTAMENTO DE COMPUTAÇÃO

BHETYNNA BRUNA KALYNNE FERREIRA CATALUNHA

IMPLEMENTAÇÃO DE AUTÔMATOS ADAPTATIVOS UTILIZANDO ORIENTAÇÃO A ASPECTOS

ALEGRE - ES

2014

BHETYNNA BRUNA KALYNNE FERREIRA CATALUNHA

IMPLEMENTAÇÃO DE AUTÔMATOS ADAPTATIVOS UTILIZANDO ORIENTAÇÃO A ASPECTOS

Trabalho de conclusão de curso apresentado ao Departamento de Computação do Centro de Ciências Agrárias da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.
Orientadora: Profª. M.Sc. Juliana Pinheiro Campos Pirovani.

ALEGRE - ES
2014

BHETYNNA BRUNA KALYNNE FERREIRA CATALUNHA

IMPLEMENTAÇÃO DE AUTÔMATOS ADAPTATIVOS UTILIZANDO ORIENTAÇÃO A ASPECTOS

Trabalho de conclusão de curso apresentado ao Departamento de Computação do Centro de Ciências Agrárias da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 30 de Julho de 2014.

BANCA EXAMINADORA

Juliana Pinheiro Campos Pirovani

Prof. M.Sc. Juliana Pinheiro Campos Pirovani
Universidade Federal do Espírito Santo
Orientadora

Edmar Hell Kampke

Prof. M.Sc. Edmar Hell Kampke
Universidade Federal do Espírito Santo

Rodrigo Freitas Silva

Prof. M.Sc. Rodrigo Freitas Silva
Universidade Federal do Espírito Santo

“Alguns homens veem as coisas como são, e dizem ‘Por quê?’ Eu sonho com as coisas que nunca foram e digo ‘Por que não?’”

Geroge Bernard Shaw

AGRADECIMENTOS

A Deus, por todo amor, cuidado e por ter me sustentado durante esses anos.

A minha família, por todo o apoio, as orações e a presença mesmo distantes.

Em especial a minha Mãe, minha Tia Sônia e a minha Avó, que nunca mediram esforços pra me ajudar a obter essa conquista, que nunca me deixaram desistir e que sempre foram meus exemplos de luta, força e superação.

Aos meus amigos distantes, que mesmo de longe estiveram comigo.

Aos meus amigos de Alegre, que foram a minha segunda família durante esses anos, que estiveram comigo nos bons e nos maus momentos, sempre prontos pra me ajudar e me apoiar. Eu não teria conseguido sem vocês.

Aos meus professores, por todo o conhecimento compartilhado.

A minha orientadora pela dedicação, empenho, pelo exemplo de responsabilidade e profissionalismo.

Essa conquista é nossa!

Muito obrigada!

RESUMO

O desenvolvimento acelerado da tecnologia nos últimos anos tem gerado sistemas cada vez mais complexos. Com isso, há uma busca constante por sistemas mais fáceis de manter, evoluir e que sejam implementados de forma mais simples e com qualidade. Com o intuito de trazer uma solução alternativa para suprir essa busca, esse trabalho propõe a implementação de duas tecnologias que, em conjunto, se mostraram eficientes: Autômatos Adaptativos e a Programação Orientada a Aspectos (POA). A habilidade da POA de inserir comportamentos vai de encontro a necessidade dos autômatos adaptativos de inserir novos comportamentos com frequência. O objetivo foi implementar exemplos de Autômatos Adaptativos com e sem Programação Orientada a Aspectos. A partir disso, foram observados alguns padrões nessas implementações e benefícios como a simplicidade, organização do código e a diminuição de acoplamento entre as classes.

Palavras-Chave: Autômatos Adaptativos, Programação Orientada a Aspectos, Teoria da Computação, Tecnologia Adaptativa.

IMPLEMENTATION OF ADAPTATIVE AUTOMATA USING ASPECT-ORIENTED

ABSTRACT

The accelerated technological development in the last several years, has generated even more complex systems. Because of it, there has been a constant search for easier systems to be develop, and maintained that can be implemented with simplicity and quality. With the goal of bringing an alternative solution to this search, this work proposes the implementation of two technologies that when working in conjunction has been proven efficient: Adaptative Automata and Aspect-Oriented Programing. The ability of aspect-oriented programming inserting behaviors meets the need of the Adaptive automata to insert new behaviors often. The objective was to implement examples of Adaptative Automation with and without Aspect-Oriented Programing. During this implementation, key patterns and benefits were observed, such as the simplicity, code organization and reduction of linkage between classes.

Key Words: Adaptative Automation, Aspect-Oriented Programming, Theory of Computation, Adaptive technology.

SUMÁRIO

LISTA DE FIGURA i
LISTA DE ABREVIATURAS E SIGLAS ii

1 INTRODUÇÃO **11**
1.1 PROBLEMA E SUA IMPORTÂNCIA 12
1.2 OBJETIVOS 14
1.2.1 Objetivo Geral 14
1.2.2 Objetivos Específicos 14

2 REVISÃO DE LITERATURA **15**
2.1 AUTÔMATOS ADAPTATIVOS 15
2.1.1 Formalização Dos Autômatos Adaptativos 16
2.1.2 Exemplos De Autômatos Adaptativos 19
2.2 PROGRAMAÇÃO ORIENTADA A ASPECTOS **27**
2.2.1 Vantagens Da POA 28
2.2.2 Desenvolvimento Com POA 30
2.3 TRABALHOS CORRELATOS **34**
2.3.1 Critérios de Inclusão e Exclusão 34
2.3.2 Fontes De Dados E Estratégias De Busca 34
2.3.3 Estudo de Identificação e Seleção 35
2.3.4 Resultado 35

3 METODOLOGIA **36**

4 RESULTADOS **38**
4.1 IMPLEMENTAÇÃO DO SIMULADOR DE PILHA 40
4.2 IMPLEMENTAÇÃO DE DEPÊNDENCIA DE CONTEXTO 42
4.3 IMPLEMENTAÇÃO DE RECUPERAÇÃO DE ERROS SIMPLES 43

5 CONCLUSÃO **46**

6 REFERÊNCIAS **47**

LISTA DE FIGURAS

Figura 1 - Simulador de Pilha 21
Figura 2 - Máquina E_1 resultante da primeira função adaptativa 22
Figura 3 - Algoritmo de recuperação de erro em AF 24
Figura 4 - Funções adaptativas pra recuperação de erro 25
Figura 5 – Reconhecedor da linguagem L 27
Figura 6 – Entrelaçamento do código 29
Figura 7 - Espalhamento do código 29
Figura 8 – Esquema de desenvolvimento com POA 30
Figura 9 – Classe conta 32
Figura 10 – Classe Autorizacao 33
Figura 11 – Aspecto contaAspect 33
Figura 12- Diagrama de classe do simulador de pilha sem POA..... 39
Figura 13 - Diagrama de classes do simulador de pilha sem POA..... 40
Figura 14 – Classe FuncaoAdaptativa 41
Figura 15– Diagrama de classes do simulador de pilha com POA 42

LISTA DE ABREVIATURAS E SIGLAS

AA Autômato Adaptativo

AF Autômatos Finitos

APE Autômato de Pilha Estruturado

DA Dispositivos Adaptativos

IDE Integrated Development Environment

LLC Linguagens Livres de Contexto

LSC Linguagens Sensíveis ao Contexto

MT Máquina de Turing

OO Orientação a Objetos POA Programação Orientada a Aspectos

RNF Requisitos Não Funcionais

TA Tecnologia Adaptativa

1 INTRODUÇÃO

Com o crescimento do mercado tecnológico nos últimos anos têm surgido softwares cada vez mais complexos e, paralelo a isso, surge também uma necessidade gradativamente mais urgente de desenvolver estes softwares com qualidade.

Existem características que podem ser úteis a sistemas com alto grau de complexidade e uma delas é a capacidade de automodificação. Essa capacidade pode ser obtida com o uso da tecnologia adaptativa (TA).

A Tecnologia Adaptativa (TA), ou simplesmente adaptatividade, refere-se a capacidade que um sistema ou abstração formal tem de se alterar, sem a interferência de qualquer agente externo, de acordo com as entradas que recebe e com seu histórico de operações (NETO, 2007).

Um exemplo de máquina com essa aptidão é o Autômato Adaptativo (AA). Os AA são dispositivos dirigidos por regras ou ações adaptativas (NETO, 2002). Esses dispositivos tem a capacidade de alterar sua própria estrutura sempre que necessário, inserindo ou removendo estados e transições, mudando assim seu comportamento em tempo de execução.

A grande vantagem da utilização desse tipo de abstração é que ela é fácil de usar, suas operações podem ser feitas de forma incremental e ela pode alterar seu próprio comportamento, sem grandes esforços, de acordo com as suas necessidades (JESUS *et al*., 2007).

Esses autômatos têm sido aplicados em diversas áreas e para diversos fins, como no processamento digital de imagens, aprendizagem de máquina, compiladores, ferramentas educacionais, inteligência artificial, etc (PISTORI, 2003).

Já em relação a qualidade dos softwares complexos, para adquirir essa qualidade é indispensável a facilidade de manutenção, leitura e compreensão. Para isso, é preciso que estes softwares tenham um código bem modularizado. Uma maneira de obter essa modularização bem definida é utilizando a Programação Orientada a Aspectos (POA).

A POA é um paradigma de programação que, segundo Campos (2009), é " utilizado para obter melhor separação de interesses, logo, tem como objetivo tornar o projeto e código de sistemas mais modulares". A sua principal função é modularizar interesses transversais que a Orientação a Objetos (OO) tem dificuldade de modularizar (BARROS, 2010).

Interesses transversais são aqueles requisitos não funcionais (RNFs) que normalmente se encontram espalhados no código. A POA modulariza esses interesses e oferece meios de inserir comportamentos. Ela possibilita adicionar código de interesses transversais nos módulos de OO (classes) sem a necessidade de modificar o código fonte original.

Baseado nessa peculiaridade, acredita-se que a POA possa ser usada para implementar os AA, facilitando a inserção de estados e transições em tempo de execução, ou seja, facilitando a chamada de funções adaptativas. É exatamente essa a proposta desse trabalho.

1.1 PROBLEMA E SUA IMPORTÂNCIA

Em Teoria da Computação são estudadas as linguagens formais, que são linguagens com sintaxe bem definida e semântica precisa (VIEIRA, 2006).

Essas linguagens possuem características distintas e podem ser reconhecidas por diferentes modelos computacionais. Tais modelos possuem peculiaridades e limitações, sendo que o mais poderoso entre eles é a máquina de Turing (MT).

A MT é um modelo matemático capaz de reconhecer linguagens recursivamente enumeráveis. Ela reconhece diversas outras linguagens, inclusive as dependentes de contexto, ou sensíveis ao contexto (LSC), que não podem ser reconhecidas por máquinas mais simples como os autômatos finitos ou de pilha. O poder desse dispositivo é apresentado por uma tese conhecida como tese de Church, ou tese de Church-Turing: "a capacidade de computação representada pela máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação" (MENEZES, 2011). Em outras palavras, a tese de Church afirma que qualquer função computável pode ser processada por uma MT.

O problema das MT, segundo Neto (2003), é que “processadores de linguagens, diretamente baseados em tais modelos, apresentam-se [..] como implementações reais de baixíssima eficiência, o que impede seu uso econômico na prática”. Por isso, há uma dificuldade na implementação de LSC (utilizadas para representar a maioria das linguagens de programação) na fase de análise sintática de compiladores. Como não há eficiência na implementação de um modelo que as reconheça puramente, é preciso usar outros artifícios. Geralmente, o que se faz é implementar reconhecedores de linguagens livres de contexto (LLC) que contenham a LSC. Porém, é necessário reinserir as características dependentes de contexto, o que é feito fora da análise sintática, através de rotinas semânticas auxiliares. Essa não é a melhor forma de resolver o problema, pois trata-se com enfoque semântico algo puramente sintático (NETO, 1993; NETO, 2003).

Uma alternativa pra resolver esse problema é o uso dos AA. Eles conseguem resolver a dependência de contexto sem o uso desses artifícios, possuem poder equivalente ao das MT e apresentam uma implementação mais eficiente (RAMOS *et al.*, 2009).

Logo, é interessante uma implementação mais simples e fácil de manter dos AA devido ao seu grande potencial de uso em Computação (NETO, 2011).

A ideia é implementar esses autômatos modularizando o código das funções adaptativas em aspectos e, posteriormente, inserir o comportamento dessas funções usando os recursos da POA. Esses recursos poderão facilitar o entendimento dos AA que tem uma formalização bastante complexa, como será apresentado posteriormente.

A possibilidade de implementação dessas tecnologias combinadas é uma hipótese válida, uma vez que a habilidade da POA de inserir comportamentos vai de encontro a necessidade dos autômatos adaptativos de inserir novos comportamentos com frequência.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Implementar Autômatos Adaptativos utilizando o paradigma de Programação Orientada a Aspectos.

1.2.2 Objetivos Específicos

- Selecionar AA ou aplicações que utilizam AA para implementação.
- Apresentar a modelagem (diagrama de estados) das aplicações com AA.
- Implementar as aplicações selecionadas utilizando POA.
- Descrever padrões identificados durante a implementação de AA com POA.
- Apresentar uma análise sobre as implementações realizadas, mostrando vantagens/ desvantagens e dificuldades dessas implementações.

2 REVISÃO DE LITERATURA

Nesta seção serão apresentados os principais conceitos envolvidos nesse trabalho, especialmente os relacionados a AA e POA. Serão apresentadas também algumas aplicações de AA e, por fim, uma revisão sistemática de literatura realizada em busca de trabalhos que tenham implementado AA com POA.

2.1 AUTÔMATOS ADAPTATIVOS

Dispositivos adaptativos (DA) são dispositivos guiados por regras, ou seja, dispositivos que têm como característica comum obedecer a um conjunto finito de regras. Nos DA, as regras não agem somente por si, mas podem receber intervenções vindas das ações adaptativas (regras adaptativas), que tem autonomia pra modificá-las. Eles são compostos por duas camadas, uma adaptativa (composta pelas ações adaptativas) e outra subjacente que é não-adaptativa (contendo as regras tradicionais) (PISTORI, 2003).

Os autômatos adaptativos são dispositivos adaptativos. É um modelo “cujo formalismo subjacente é algum autômato” (Ramos *et al.,* 2009), podendo esse ser finito, de pilha ou de pilha estruturado. Um autômato de pilha estruturado (APE), segundo Neto e Rocha (2014) é “um conjunto de submáquinas finitas agrupadas e com possibilidade de efetuar chamadas e retornos entre si, através do uso de pilha”.

O APE é composto por um conjunto de autômatos subjacentes (submáquinas), onde cada um deles tem a responsabilidade de reconhecer uma determinada parte da cadeia de entrada em análise. Eles possuem uma pilha, que diferentemente dos autômatos de pilha comuns, é restrita somente às transições de chamadas e retornos de submáquinas.

Usualmente, associa-se os APE como o modelo da camada subjacente dos AA, pois historicamente o primeiro autômato adaptativo era um autômato de pilha estruturado adaptativo. No entanto, como já foi dito, qualquer autômato pode compor essa camada.

Os AA podem ser definidos como um modelo computacional que tem a capacidade de alterar (inserindo, removendo ou modificando transições e estados), sem

qualquer interferência, seu comportamento em tempo de execução. Essa capacidade de adaptação é o grande poder desse dispositivo (ZUFFO; PISTORI, 2004). Além disso, é também essa característica que torna os AA capazes de reconhecer as LSC (NETO, 2003).

Inicialmente, o AA tem uma topologia simples e fixa, representada por uma única máquina, a máquina inicial E_0. Cada alteração de topologia caracteriza uma nova máquina. Assim, após a execução da primeira ação adaptativa ele se torna uma máquina E_1, depois da segunda E_2 e na n-ésima alteração ele se transforma na máquina E_n.

2.1.1 Formalização Dos Autômatos Adaptativos

A formalização de AA apresentada nesta seção é a que foi definida por Neto (2003).

As regras de transição dos AA são do tipo: (γg , e , sα) : A → (γg' , e' , s'α), B. O que se encontra antes da seta é a situação do AA antes da transição e o que está depois dela é a situação após a aplicação da transição, sendo:

- γ é o conteúdo da pilha que não foi analisado. O símbolo g é o conteúdo do topo da pilha no momento anterior à aplicação da transição, e g' o topo da pilha logo após a aplicação da regra (g, g' $\in \gamma$.). Quando g $\neq \varepsilon$ (sendo ε o símbolo vazio), ele é desempilhado; quando é omitido, indica que a transição será realizada independente do conteúdo da pilha. Quando g'$\neq \varepsilon$, ele é empilhado; quando é omitido, indica que nada será empilhado devido a aplicação da transição. Se g = g' e diferente de ε, indica que a transição será realizada se o topo da pilha for g.

- **e** é o estado corrente e e' o estado alcançado após execução da regra;

- α representa a cadeia de entrada que ainda não foi analisada. Se s $\in \alpha$ aparece na regra, ele será consumido, ou seja, retirado da cadeia de entrada. Quando s' $\in \alpha$ está presente na regra, ele é inserido na cadeia de entrada. É importante salientar que s, quando acompanhado de α, indica que a transição acontece se, e somente se, o autômato estiver no estado e, com g no topo da pilha e recebendo o estímulo s (lendo s da entrada). O símbolo s pode ser

vazio (ε) ou simplesmente omitido na regra. Isso significa que a transição ocorre sem levar em conta o estímulo recebido;

- **A** (opcional) representa uma ação adaptativa a ser aplicada antes que a transição ocorra de fato, ou seja, antes que a configuração após a seta seja alcançada. Geralmente, essa função será chamada de função anterior.

- $\rightarrow$ representa a transição em si, ou seja, a mudança de configuração da máquina.

- **B** (opcional), também conhecida como função posterior, representa uma ação adaptativa a ser aplicada após a execução da transição.

Em alguns casos, partes da estrutura da regra apresentada anteriormente são omitidas. Isso simplifica o modelo, uma vez que certas linguagens não demandam todos esses recursos no seu reconhecimento. Por exemplo, quando o AA for um reconhecedor de linguagens regulares, as regras que o compõem não necessitam nem dos símbolos de pilha, nem dos símbolos de ações adaptativas. É usada uma regra do tipo: (e , s α) : $\rightarrow$ (e' , α) que pode ser entendida como: estando no estado e, lendo s, desloca-se pra e' consumindo s.

Vale ressaltar que a pilha só compõe o AA quando sua camada subjacente é composta por um APE. Em outros casos essa pilha e todos os símbolos que a representam não aparecem.

Dado o conteúdo da pilha, um estado corrente e uma palavra a ser lida, as regras do modelo são consultadas em busca de alguma aplicável a situação. Nesse momento, três coisas podem acontecer:

1. Ter uma única regra que se aplica ao caso, sendo ela aplicada e seguindo a execução normalmente (transição determinística).

2. Ter mais de uma regra aplicável, o que caracteriza o não-determinismo. Quando isso ocorre, todas as possibilidades são analisadas para verificar se uma delas leva a aceitação da cadeia. Se uma computação termina em um estado final, a cadeia é aceita e todas as execuções ainda em paralelo (se existirem), são interrompidas. Se todas rejeitarem, a cadeia é rejeitada.

3. Quando nenhuma regra se aplica. Nesse caso, a palavra é rejeitada, já que ainda existem símbolos a serem consumidos.

A situação inicial de um AA é caracterizada por: pilha vazia (indicada por Z_0), estado inicial $e_0 \in E_0$ e a cadeia de entrada ainda completa. A situação final é: pilha vazia (Z_0), palavra consumida (ε) e estado final $e_f \in F$ (conjunto de estados finais contendo todos os estados finais das submáquinas).

As funções adaptativas podem ser vistas como uma função de uma linguagem de programação. Elas são responsáveis por executar uma tarefa, podendo retornar algo e até mesmo sendo parametrizadas.

Uma função adaptativa é composta por:

- Um nome N que a identifique;
- Uma lista de parâmetros que, inicialmente não possuem qualquer valor, e permanecem indefinidos até que alguma ação adaptativa os preencha.
- Uma lista de variáveis (opcional) que serve para referenciar (por nome) valores que não poderão ser alterados durante a execução da função e que são passados a elas por chamadas às funções adaptativas;
- Uma lista de geradores que são preenchidos no início da execução da função com valores novos (ainda não utilizados na definição do AA). Esses permanecem inalterados até que a execução do AA finalize. São eles que atribuem nomes aos novos estados inseridos a partir de uma ação adaptativa de inserção (definida a seguir);
- Uma lista de ações adaptativas que têm ordem de precedência na execução. São executadas as ações de consulta, de remoção e de inserção, exatamente nessa ordem. A definição de cada uma é dada abaixo:
    - **Ações de Consulta** são representadas por um '?', seguido de uma transição entre colchetes:

      $?[(\gamma g , e , s \alpha) : A , \rightarrow (\gamma g' , e' , s'\alpha) , B]$

Essas ações tem por objetivo encontrar regras de transição que obedeçam a estrutura da regra que é dada entre colchetes. Se alguma regra desse tipo é encontrada, as variáveis indefinidas que compõem a regra representada acima são preenchidas pelos valores das regras encontradas. Caso isso não aconteça, as variáveis seguem indefinidas. Esse tipo de ação não altera a topologia do AA;

- **Ações de Inserção** são representadas por um '+' antes da transição:

  $+ [(\gamma g , e , s \alpha) : A , \rightarrow (\gamma g' , e' , s'\alpha)]$

  Essas ações inserem novas regras no AA se todos os seus componentes são definidos. Elas podem conter novos estados (criados pelos geradores) e transições, alterando assim a estrutura do AA. Caso a regra criada já exista no conjunto de regras, a ação não a duplica.

- **Ações de Remoção** são representadas por um '-' antes da transição:

  $- [(\gamma g , e , s \alpha) : A , \rightarrow (\gamma g' , e' , s'\alpha)]$

  Inicialmente, tem comportamento idêntico a uma ação de consulta, buscando regras que sigam o padrão especificado entre colchetes. Porém, ela elimina do AA todas as regras encontradas nessa busca.

Chamadas às funções adaptativas terão a seguinte estrutura: $N(\Phi 1, \Phi 2, \Phi 3, ..., \Phi m)$ onde Φ é o símbolo escolhido pra representar um argumento.

2.1.2 Exemplos De Autômatos Adaptativos

Nesta seção serão apresentados alguns exemplos de AA.

2.1.2.1 Simulador De Pilha

Proposto por Neto(1993), esse simulador utiliza autômatos adaptativos para reconhecer uma palíndrome do tipo: $(^n\beta)^n$. Ou seja, β, (β), ((β)) são exemplos de cadeias reconhecidas por esse modelo.

As regras iniciais da máquina S são:

S = {

(1 , " β ") : → 4

(2 , " β ") : → 3

(3 , ")") : → 4

(1 , "(") : → 2 , $\mathcal{A}$(2 , 3 , 1) }

Esse AA não utiliza pilha, por isso não há nas regras acima nenhum símbolo que remeta a ela. Nesse caso, $\mathcal{A}$ é equivalente ao B da formalização descrita, ou seja, $\mathcal{A}$ é uma ação adaptativa posterior.

Pode-se compreender a regra (1," β "): → 4 como: 1 é o estado (e) corrente, " β " é o símbolo (s) a ser consumido e 4 é o estado (e') resultante da aplicação da regra. As outras regras tem comportamento análogo, com exceção da regra (1 , "(") : → 2 , $\mathcal{A}$ (2 , 3 , 1), que chama a função adaptativa de nome $\mathcal{A}$ passando os argumentos 2, 3 e 1.

A função adaptativa da aplicação é:

$\mathcal{A}$(i , j , n) = { k*, m* :

+ [(k, "β") : → m]

+ [(m, ")") : → j]

+ [(i, "(") : → k, $\mathcal{A}$(k , m , i)]

- [(n, "(") : → i, $\mathcal{A}$(i , j , n)]

+ [(n, "("): → i] }

O caractere (*) após **k** e **m** indica que eles são os geradores da função.

O diagrama de estados da máquina proposta pode ser visto na Figura 1.

Neto (1993) representa, graficamente, as ações adaptativas anteriores e posteriores a uma função com um ponto. Se esse ponto aparece antes do nome da função, como é o caso do exemplo, significa que a chamada à função adaptativa ocorre depois de executada a transição. Caso o ponto apareça depois, significa que a

função será chamada antes da transição ocorrer. Ou seja, nessa aplicação a chamada à função se dá logo após a ocorrência da transição.

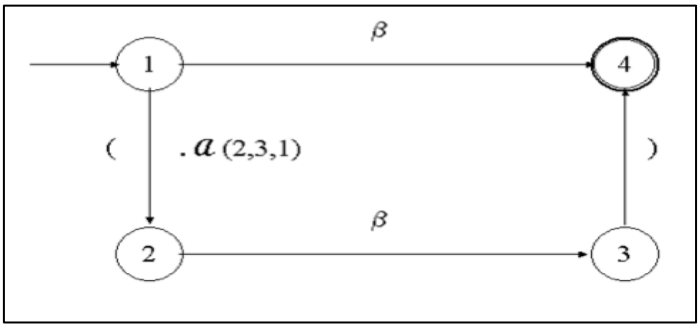

Figura 1 - Simulador de Pilha

Fonte: Neto (1993)

Veja a descrição do funcionamento do AA para a cadeia de entrada “((β))”:

1. Iniciando no estado 1 e lendo o primeiro símbolo “(” da palavra, observamos que a regra (1 , "(") :→ 2 , $\mathcal{A}$(2, 3,1) deve ser aplicada. Ela leva a execução pro estado 2, consome o símbolo “(”e chama a função adaptativa;
2. Percebe-se que $\mathcal{A}$ é parametrizada e chama a função passando os parâmetros 2, 3 e 1, nessa ordem. As seguintes atribuições são feitas: i = 2, j = 3 e n = 1. Os geradores k e m geram automaticamente valores inéditos no modelo, que mais tarde serão os identificadores dos novos estados. Eles podem assumir, por exemplo, os valores 5 e 6, respectivamente. A função com todos os valores definidos é:

   $\mathcal{A}$(2 , 3 , 1) = { 5*, 6* :

   + [(5, “β”) : → 6]

   + [(6 “)”) : → 3]

   + [(2, “(”) : → 5, $\mathcal{A}$(5 , 6 , 2)]

   - [(1, “(”) : → 2, $\mathcal{A}$(2 , 3 , 1)]

   + [(1, “(”): → 2] } ;

3. Com todos os valores da função definidos, pode-se então executá-la. Seguindo a ordem de precedência das ações, a primeira a ser realizada é a de eliminação. Substituindo os valores definidos no passo 2, a ação fica: -[(1,

"(") : → 2, $\mathscr{A}$(2 , 3 , 1)]. A busca por uma regra desse tipo será feita e ela será excluída.

4. A seguir, todas as ações adaptativas de inserção serão realizadas. Como exemplo, a ação de inserção +[(5, "β") : → 6] insere os estados 5 e 6 que não existiam e insere a transição entre eles rotulada por β. Segue a máquina E_1 obtida após execução da função adaptativa:

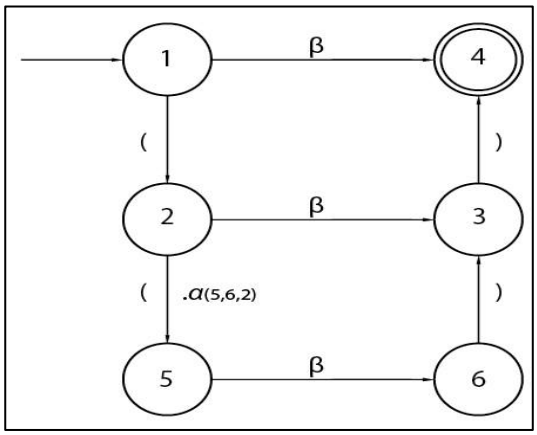

Figura 2 - Máquina E_1 resultante da execução da função adaptativa

5. Com essa nova máquina E_1, o processo de reconhecimento da cadeia, que agora contém os símbolos "(β))" continua. A partir do estado 2, a computação segue passando pelos estados 5, 6, 3 e 4 consumindo respectivamente os símbolos "(", "β", ")" e ")" e reconhecendo a palavra. Observe que após a transição entre os estados 2 e 5, a função adaptativa será invocada novamente e a topologia do AA será mais uma vez modificada, de forma análoga ao que ocorreu no passo 2. No entanto, a topologia anterior a essa chamada de função já atende a necessidade de reconhecimento da cadeia proposta.

Apesar da simplicidade dessa aplicação, ela se mostra muito eficiente pra demonstrar o comportamento dos AA, mostrando inclusive como ele é capaz de funcionar sem uma pilha.

2.1.2.2 AA Em Compiladores

Por reconhecer linguagens dependentes de contexto, o AA é amplamente aplicado na área de compiladores. Isso porque as linguagens de programação geralmente

são dependentes de contexto. Um bom exemplo disso é que, na maioria delas, não é possível realizar operações com variáveis que não tenham sido declaradas. Ou seja, entende-se que o uso da variável depende da sua declaração.

Pistori (2003) apresenta, entre outras aplicações de AA, técnicas de recuperação de erros sintáticos em compiladores.

A implementação de uma técnica de recuperação de erros em um compilador é fundamental, uma vez que quem o utiliza são os seres humanos. Na escrita de um código, eles podem cometer erros simples, como digitar algum termo errado ou esquecer um ponto e vírgula. Quando o compilador processa um código com erro, ele não é capaz de prosseguir com a compilação e a recuperação de erros é realizada.

Existem várias técnicas de recuperação de erros, algumas que simplesmente interrompem a compilação quando o erro é detectado, outras que tentam corrigir o código fonte com erro e outras ainda que tentam continuar o processo mesmo na presença do erro, se recuperando automaticamente. Essa última é a mais almejada e estudada.

Pistori (2003) propôs uma recuperação de erros simples, primeiramente utilizando autômatos finitos (AF). A detecção do erro corresponde a alcançar um estado a partir do qual não existe transição para o próximo símbolo. Normalmente, esses erros ocorrem por omissão, inserção ou substituição de um símbolo. Por isso, o AF trabalha tentando excluir, inserir ou substituir o símbolo certo através de manipulação de transições e estados. A aplicação demonstrada por ele é simples, mas funciona e só precisa ser adaptada pra problemas mais complexos. Segue a aplicação proposta utilizando AF:

Definição: Dado um AF M = (Q, Σ, Q_0, F, δ), onde Q é o conjunto de estados, Σ o conjunto de símbolos do alfabeto de entrada, Q_0 o estado inicial, F o conjunto de estados finais e δ a função de transição de M.

São definidas as funções:

- *First(q):* conjunto de estados alcançáveis a partir do estado q após a leitura de um símbolo da cadeia de entrada.

- *Second*(q): conjunto de estados alcançáveis a partir do estado q, após a leitura de dois símbolos da cadeia de entrada.
- *firstSymbols*(q): conjunto de símbolos que podem ser lidos a partir do estado q.
- θ: denota o conjunto Σ − *firstSymbols*(*q*)

Dadas as definições, o algoritmo é apresentado na Figura 3.

Recuperação de erros simples em AF

entrada: AF $M=(Q,\Sigma,q_0,F,\delta)$
saída: M com recuperação de erro
para cada $q \in Q$ **faça**
  $Q \leftarrow Q \cup \{e_1, e_2\}$ {Adiciona dois novos estados}
  Adicione uma transição de erro de q para e_1
  para cada $c \in \theta$ **faça**
    $\delta \leftarrow \delta \cup (e_1, c, e_2)$
  fim para
  para cada $q_s \in second(q)$ **faça**
    Seja $b \in \Sigma$ o símbolo que garante a presença de q_s em $second(q)$
    $\delta \leftarrow \delta \cup (e_1, b, q_s)$
    $\delta \leftarrow \delta \cup (e_2, b, q_s)$
  fim para
  para cada$q_f \in first(q)$ **faça**
    Seja $a \in \Sigma$ o símbolo que garante a presença de $q_f$em $first(q)$
    $\delta \leftarrow \delta \cup (e_2, a, q_f)$
  fim para
  se q é um estado final **então**
    Faça de e_2 um estado final
  fim se
fim para

Figura 3 - Algoritmo de recuperação de erro em AF
Fonte: Adaptado de Pistori (2003)

Pistori (2003) apresentou também a técnica com AA. Nesse caso, o AA poderia usar o AF antes proposto como sua máquina subjacente. Ou seja, quando ocorre um problema no reconhecimento da linguagem, o AA chama uma função adaptativa que altera sua topologia (através da inserção de transições de erro) pra tratar o erro. Essa modelagem tem a vantagem de ser mais simples e ocupar menos espaço em memória do que a modelagem com AF, uma vez que só adicionaria momentaneamente (enquanto necessário) os estados e as transições pra recuperação.

No caso da solução com AA, tem-se duas funções adaptativas: $\mathcal{E}$ e $\mathcal{E}'$. A primeira é responsável por modificar o autômato pra que ele possa se recuperar de erros e a segunda é responsável por remover as transições de erros criadas pela primeira função, depois que tenha acontecido a recuperação.

A Figura 4 representa graficamente a aplicação das funções adaptativas, onde δ representa a transição de erro.

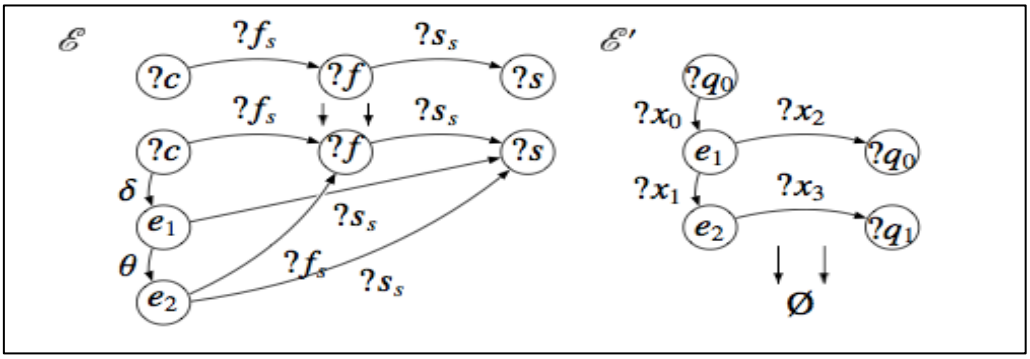

Figura 4 - Funções adaptativas pra recuperação de erro

Fonte: Pistori (2003)

Para melhor entendimento do funcionamento da aplicação, será apresentado um exemplo a seguir.

Considere a linguagem LG representada pela expressão regular ab*c, (palavras que começam com “a”, terminam com “c” e tenham 0 ou mais “b”s entre “a” e “c”), sendo Σ = {“a”, “b”, “c”}. O reconhecedor de LG é:

M_0 = {

(1,"a") : -> 2

(2,"b") : -> 2

(2,"c") : -> 3}

A cadeia “bc” não pertence a LG. Isso significa que, ao receber essa cadeia de entrada, o compilador não consegue terminar a compilação. Então, a recuperação de erros é acionada. Seguindo o algoritmo da Figura 3, a seguinte máquina é gerada a partir da detecção do erro:

M_1 = {

(1,"a") : -> 2

(2,"b") : -> 2

(2,"c") : -> 3

(4,"b") : -> 5

(4,"c") : -> 5

(4,"b") : -> 2

(5,"b") : -> 2

(4,"c") : -> 3

(5,"c") : -> 3

(5,"a") : -> 2 }

O estado 4 é equivalente a e_1 e o estado 5 a e_2. Com isso, a compilação chega ao fim e a entrada é rejeitada uma vez que não pertence a LG.

2.1.2.3 AA Para Reconhecer Linguagem Sensível Ao Contexto

Como apresentado na seção 1.1, uma característica importante do AA é o poder de reconhecimento das LSC. Taniwaki e Neto (2001) apresentam um exemplo prático de um reconhecedor de LSC, apresentada a seguir.

A linguagem gerada pela gramática L abaixo inclui expressões utilizando as variáveis “a” e “b”, se elas tiverem sido previamente declaradas:

L = {

PROG → (a | b | a,b | b,a) : S_2

S_2 → (a | b | <S_2>) ((+ | - | * | /) (a | b |<S_2>))* }

Na Figura 5, pode ser visto o reconhecedor da linguagem gerada por L. As linhas que aparecem pontilhadas em S_2 não existem inicialmente no autômato. Quando “a” é lido em PROG, a função adaptativa A é chamada e insere as transições (1, “a” , 2) e (3, “a”, 2) em S_2. De forma análoga, as transições (1, “b” , 2) e (3, “b”, 2) são inseridas em S_2 quando acontece a leitura do símbolo “b” em PROG. Isso garante que as expressões só serão reconhecidas se as variáveis que a compõem tiverem sido declaradas, o que indica a dependência de contexto da linguagem gerada por L.

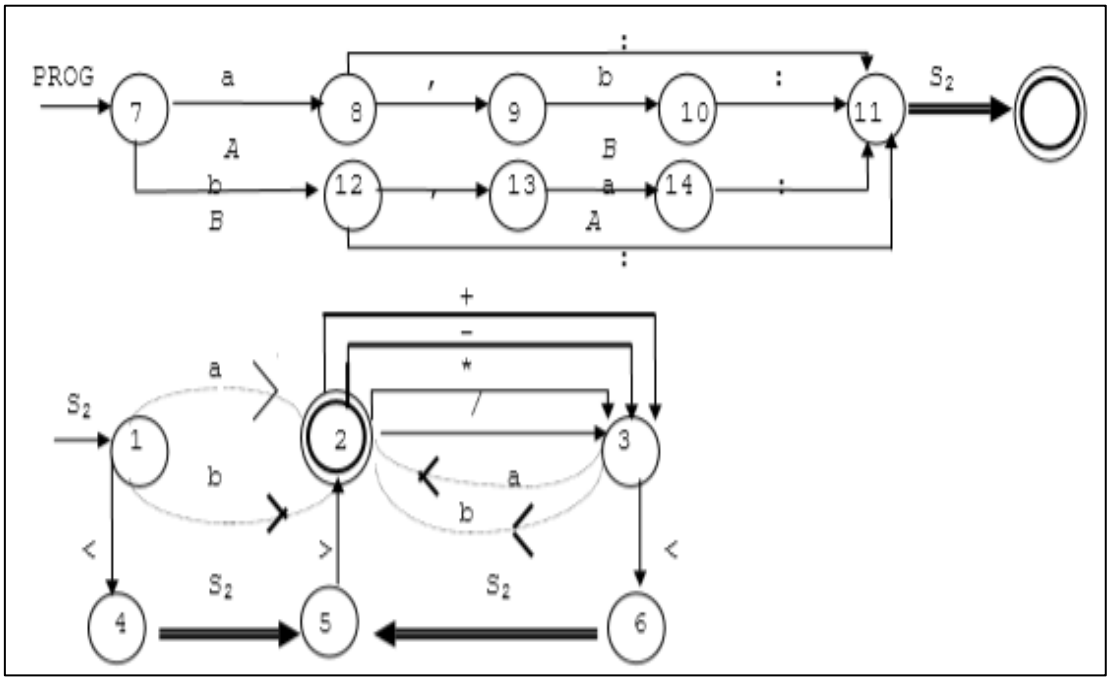

Figura 5 – Reconhecedor da linguagem gerada por L

Fonte: Taniwaki e Neto (2001)

2.2 PROGRAMAÇÃO ORIENTADA A ASPECTOS

A programação orientada a aspectos (POA) surgiu a partir de pesquisas de universidades e empresas para romper os limites da orientação a objetos (OO). Limites esses que muitas vezes impedem que o desenvolvimento dos sistemas possam ser simplificados e resultam em softwares pouco gerenciáveis e com baixa flexibilidade (RESENDE e SILVA, 2005).

A POA tem como objetivo a separação de interesses. Quando se fala em separação de interesses, não quer dizer que a OO seja incapaz de fazê-lo. No entanto, ela encontra dificuldade em separá-los quando eles se encaixam naturalmente em mais

de um módulo do sistema ou em módulos fortemente relacionados (SILVEIRA, 2005).

Soares *et al.* (2014) explica que uma propriedade de um sistema pode ser implementada de duas formas:

1. Como um **componente**, quando ela pode ser encapsulada em um procedimento generalizado. Nesse caso, essa propriedade tende a ser uma única unidade da decomposição funcional do sistema. Exemplos de componentes são: métodos, objetos, procedimentos, etc;
2. Como um **aspecto**, quando ela não pode ser vista como uma única unidade de decomposição funcional do sistema, mas sim como uma propriedade distribuída ou presente em várias outras contribuindo direta ou indiretamente à semântica dessas. Exemplos de aspectos são restrições de segurança, de acesso, autenticação, tratamento de erros e exceções, persistência, etc. Eles são chamados também de interesses transversais do sistema.

A partir dessa visão sobre a implementação de uma propriedade do sistema, Soares et al. (2014) define POA como uma " técnica que pode oferecer suporte para o programador na tarefa de separar claramente os componentes dos aspectos, os componentes entre si e os aspectos entre si, utilizando-se de mecanismos que permitam a abstração e composição destas, produzindo o sistema desejado".

A POA é utilizada em conjunto com outras metodologias, como OO, que conseguem dar o suporte desejado a separação de componentes.

2.2.1 Vantagens Da POA

A baixa modularidade do sistema gerada por tecnologias que não tem suporte pra lidar com aspectos e componentes da maneira correta, pode acarretar em uma série de problemas, dentre eles o espalhamento e o entrelaçamento de código. De acordo com Resende e Silva (2005):

- O **entrelaçamento** se dá quando um só módulo do sistema possui código de múltiplos interesses, como pode ser visto na Figura 6.

O entrelaçamento dificulta a manutenção, pois uma vez que se faça necessário qualquer alteração em uma classe que tenha alto entrelaçamento, o impacto gerado no sistema é grande.

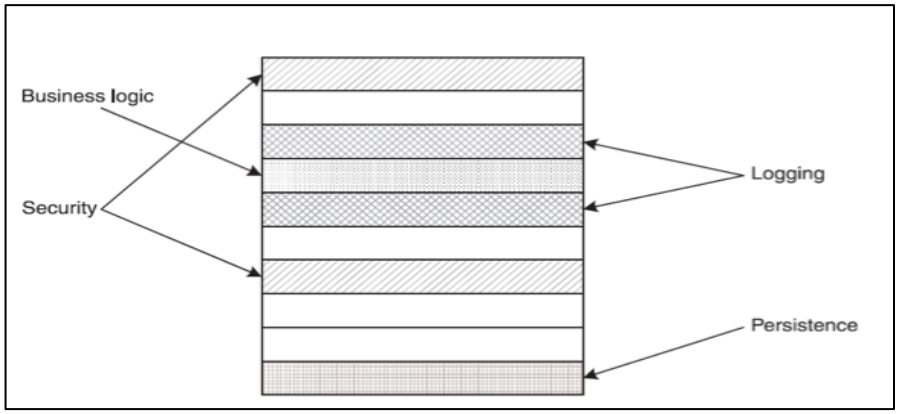

Figura 6 – Entrelaçamento do código

Fonte: Laddad (2003)

- O **espalhamento** se dá quando um código está espalhado (repetido) em diversas classes. Isso gera dificuldade de manutenção pois, para realizar qualquer alteração no sistema, será preciso buscar todos os lugares onde aquele código está e alterá-lo também. O espalhamento é ilustrado abaixo:

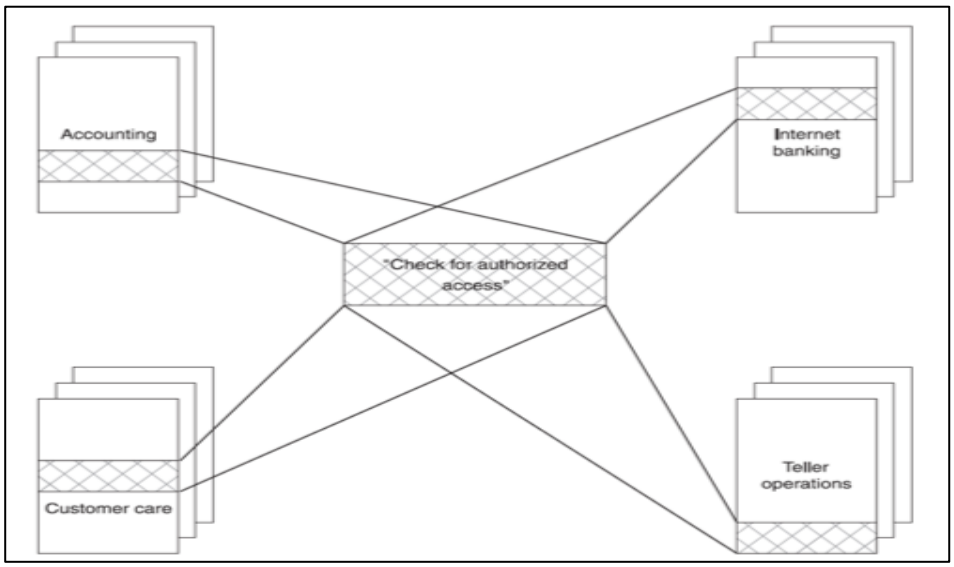

Figura 7 - Espalhamento do código

Fonte: Laddad (2003)

A POA surge como alternativa para acabar com ambos os problemas. Com a sua utilização, retira-se das classes todo o trabalho que não é delas e esse trabalho é

inserido nos aspectos. A partir disso, surgem diversos benefícios como a facilidade de reutilização da classe original. A POA também possibilita evitar a repetição de código que acontece na verificação de acesso da figura 7. O código relacionado a essa verificação seria implementado uma única vez, em um aspecto, que seria inserido nas classes onde e quando se fizer necessário.

2.2.2 Desenvolvimento Com POA

Segundo Soares et al. (2014), o desenvolvimento com POA, geralmente engloba alguns itens. São eles:

- Uma linguagem que implementa a separação de componentes;
- Uma ou mais linguagens orientadas a aspectos;
- Um *aspect weaver* ou compilador de aspectos pra integrar os diversos interesses implementados usando as duas linguagens;
- Um programa escrito em linguagem de componentes;
- Um ou mais programas em linguagens orientadas a aspectos.

Abaixo, a Figura 8 representa o desenvolvimento com POA:

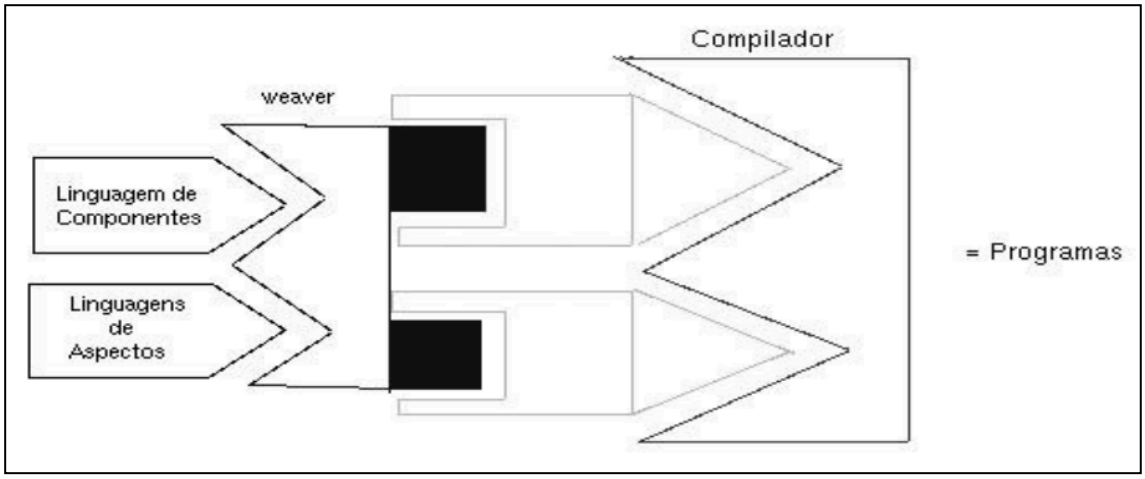

Figura 8 – Esquema de desenvolvimento com POA

Fonte: Soares et al. (2014)

Para desenvolver uma aplicação com POA é preciso definir qual linguagem de componentes usar. Essa linguagem pode ser orientada a objetos como Java, ou até mesmo uma estruturada como C.

Uma ou mais linguagens orientadas a Aspectos também devem ser escolhidas. Campos (2009) cita algumas como *AspecJ* , *HyperJ* , *AspectC* e *AspectC++* e até um compilador independente de aspectos chamado *Weave.Net* . No entanto, afirma que a mais citada na literatura é a *AspecJ* .

A combinação do programa escrito em linguagem de componentes com o programa escrito em linguagem orientada a aspectos é feita pelo ***aspect weaver***. De acordo com Piveta (2001): **"**a função do combinador de aspectos é processar a linguagem de aspectos e a de componentes, compondo essas linguagens corretamente a fim de produzir a operação geral desejada". Depois de combinadas as linguagens elas são compiladas e o programa final é gerado.

A linguagem de aspectos *AspecJ* foi estendida do Java e pode implementar interesses transversais de forma estática ou dinâmica. A transversalidade estática permite inserir comportamentos que alteram a estrutura estática do sistema, como adicionar métodos e atributos a classes, declarar erros e enfraquecer exceções. Já a transversalidade dinâmica permite inserir comportamentos em tempo de execução. Esses comportamentos podem ser inseridos em qualquer ponto bem definido do programa como antes ou depois da execução de um método (TIRELO, 2004).

É a transversatilidade dinâmica que permite a inserção do código relacionado a interesses transversais nos demais módulos do programa. Para compor esses diversos interesses do sistema, *AspectJ* utiliza os seguintes recursos (Resende e Silva, 2005):

- ***Joinpoint*** *(*ponto de junção ou ponto de combinação): ponto bem definido no fluxo de execução do programa. São esses pontos que possibilitam definir e identificar locais de interceptação no programa onde podem ser inseridos comportamentos de interesses transversais. Pontos de junção podem ser chamadas e execução de métodos, de construtores, execução de exceções, etc.

- ***Pointcuts*** (conjuntos de junção): são os responsáveis por declarar os *joinpoints*, ou seja, são eles quem declaram os pontos onde posteriormente poderá ser inserido algum comportamento. Pode-se visualizar um conjunto de junção como sendo uma variável que desempenha o papel de armazenar

todas as assinaturas de métodos que se deseja interceptar com o intuito de inserir novas chamadas ou inserir *advices*.

- ***Advices (***regras de junção): implementam o comportamento a ser inserido, o código dos interesses transversais. Podem ser vistos como métodos que serão executados no momento em que um ponto de junção for identificado. Eles podem entrar em ação de três formas: imediatamente antes (***before***) de um *joinpoint*, imediatamente depois (***after***) de um *joinpoint* ou podem alterar o fluxo de execução do programa (***around***). O *around* é executado no lugar do *joinpoint* interceptando-o e tomando pra si o controle do fluxo em questão, podendo até mesmo impedir que o método anteriormente chamado seja executado. Isso concede ao programador total poder sobre o que será ou não executado naquele instante.

- ***Aspects*** (aspectos): responsável por encapsular os *joinpoints*, *pointcuts* e *advices*, assim como as classes na OO encapsulam métodos e atributos.

Para entender como funciona na prática o conceito de aspecto será apresentado um exemplo nas Figuras 9, 10 e 11.

```
1    public class Conta {
2
3    private int numero;
4    private float saldo;
5    private String nome;
6
7    public Conta(int numero, float saldo, String nome){…}
8
9    public int getNumero(){…}
10   public void setNumero(int numConta){…}
11   public float getSaldo(){…}
12   public void setSaldo(float saldoConta){…}
13   public String getNome(){…}
14   public void setNome(String nomeTitularConta){…}
15
16   public void depositar(float valor) {…}
17
18   public void verificarSaldo(){…}
19   }
20   ...
```

Figura 9 – Classe conta
Fonte: Campos (2009)

```
1    public class Autorizacao {
2    …
3
4    public static boolean estaAutorizado(){…}
5
6    …
7    }
```

Figura 10 – Classe Autorizacao
Fonte: Campos (2009)

```
1    public aspect ContaAspect {
2
3    pointcut RealizaDeposito(): execution(public void
     Conta.depositar(float));
4    pointcut VerificaAutorizacao(): call(public void Conta.*(..));
5
6    after(): RealizaDeposito(){
7      System.out.println("Transação efetuada: Realização de depósito.");
8
9    }
10
11   void around(): VerificaAutorizacao(){
12     if(Autorizacao.estaAutorizado()){
13       System.out.println("Usuário autorizado a realizar a transação: " +
     thisJoinPoint.getSignature().getName());
14       proceed();
15   }
16
17   else
18     System.out.println("Você não possui permissão para realizar essa
     transação");
19    }
}
```

Figura 11 – Aspecto contaAspect

Fonte: Campos (2009)

Os métodos da classe Conta só podem ser realizados mediante a autorização da classe Autorizacao. Sem a POA, o método estaAutorizado da classe Autorização teria que ser chamado dentro de todos os métodos da classe Conta. No entanto, essa verificação de autorização pode ser inserida nessa classe pelo aspecto ContaAspect.

Observando ContaAspect, pode-se perceber que existem dois *pointcuts*: um, de nome RealizaDeposito, que contém um *joinpoint* para capturar a execução (*execution*) do método depositar da classe Conta. O outro, de nome VerificaAutorizacão, contém um ponto de junção que captura a chamada (*call*) de todos os métodos da classe Conta.

Os *advices* também são notados no exemplo. Existem dois deles: o primeiro (linhas 6-9) imprime uma mensagem na tela do usuário após (*after*) o *pointcut* RealizaDeposito, ou seja, após a execução do método depositar da classe Conta. O outro (linhas 11-19) interrompe as chamadas dos métodos da classe Conta e verifica se o usuário está autorizado; se sim, ele permite que a execução do método chamado prossiga normalmente (*proceed*), caso contrário ele informa ao usuário que ele não possui autorização para tal.

2.3 TRABALHOS CORRELATOS

Foi realizada uma revisão sistemática de literatura que tem por objetivo fazer uma pesquisa não tendenciosa sobre um determinado assunto e mostrar claramente como foi realizada a pesquisa. Ela é utilizada para comprovar os resultados obtidos e permitir que qualquer pesquisador faça a mesma busca e encontre os mesmos resultados.

O objetivo dessa revisão foi investigar trabalhos e/ou implementações que tenham usado a POA para implementar AA. Essa revisão foi baseada no processo proposto por Kitchenham (2004).

A principal questão a ser investigada é: existe alguma implementação de autômatos adaptativos utilizando a orientação a aspectos?

2.3.1 Critérios de Inclusão e Exclusão

Foram analisados diversos artigos em busca de encontrar algum que utilizasse a POA para implementar AA.

O objetivo seria incluir todos os artigos que apresentassem uma implementação de autômatos adaptativos com orientação a aspectos e excluir os demais.

2.3.2 Fontes De Dados E Estratégias De Busca

É sabido que termos diferentes são usados na literatura pra se referir ao mesmo assunto. Por isso, para buscar artigos, vários termos foram inseridos.

A seguir é apresentado um quadro contendo as cadeias de palavras utilizadas em cada pesquisa, separadas por buscador.

Nome das ferramentas de busca	**Cadeias de palavras pesquisadas**
IEEEXplore, ACM Digital Library, Springer, Wiley, Science Direct	*adaptative automata with Aspect orientation, implementation adaptative automata, aspects oriented programming in adaptative automata applications*
Scielo, Acervo da Capes, Scholar	uso de programação orientada a aspectos em autômatos adaptativos, autômatos adaptativos com orientação a aspectos, implementação de autômatos adaptativos

Quadro 1 – Palavras Utilizadas Para Pesquisa Por Ferramenta De Busca

2.3.3 Estudo de Identificação e Seleção

A pesquisa foi realizada no mês de janeiro de 2014 pela autora deste trabalho. Houve uma análise de resumos e palavras chaves nos artigos encontrados em busca de algo que remetesse ao que estava sendo procurado.

Nessa análise foi encontrado um único trabalho relevante escrito por Casachi e Camolesi (2012). Esse trabalho é uma proposta de uso de POA no desenvolvimento de aplicações que utilizam conceitos de TA. Ao estudar o texto, foi observado que ele não era exatamente o que estava sendo procurado pois só trata da possibilidade de implementar TA com POA, mas nem sequer menciona AA, nem apresenta alguma implementação. Por isso, pelos critérios de inclusão e exclusão apresentados anteriormente, esse artigo também não foi considerado.

2.3.4 Resultado

A partir da investigação realizada pode-se concluir que: utilizando as principais palavras-chave relacionadas ao assunto, não foi encontrado na literatura nenhum trabalho que apresente a implementação de AA com POA.

3 METODOLOGIA

Inicialmente foi realizada uma revisão bibliográfica com o intuito de aprofundar os conhecimentos das duas tecnologias (AA e POA), conhecer trabalhos correlatos e aplicações de AA.

Também foi feita uma revisão sistemática de literatura baseada no que foi proposto por Kitchenham (2004), em busca de implementações de AA que tenham sido feitas com o auxílio de POA. Essa revisão foi realizada porque inicialmente não foram encontradas implementações de AA utilizando recursos de POA.

Posteriormente, foram escolhidos os AA a serem implementados. Para realizar essa escolha foram analisadas as características das aplicações. O objetivo foi selecionar um exemplo mais simples com uma única função adaptativa, um exemplo com duas funções adaptativas e um APE como máquina subjacente, e um exemplo um pouco mais elaborado com função adaptativa que pode ser chamada a qualquer momento. Os AA escolhidos para implementação foram os apresentados na sessão 2.1.2. O simulador de pilha foi a primeira implementação. O exemplo de dependência de contexto a segunda, já que tinha o APE S_2 e contava com duas funções adaptativas: A e B. O terceiro exemplo foi o AA pra recuperação de erros simples, uma vez que tinha uma função adaptativa que poderia ser chamada a qualquer momento durante a execução do AA e por ser aplicável a qualquer linguagem. Nessa última aplicação só foi implementada a função $\mathcal{E}$, responsável por inserir as transições de erro.

Como todos os exemplos escolhidos já tinham seus respectivos diagramas de estados, só foi necessário projetar os diagramas de classes para cada implementação.

Em seguida, os três AA foram implementados com e sem POA. A linguagem utilizada para implementação foi Java. Para as implementações com POA também foi utilizada a linguagem *AspectJ*. A IDE escolhida para o desenvolvimento foi o Eclipse por conter o *plugin* mais recente do *AspecJ*.

Por fim, foi observado se a aplicação de POA na implementação de AA foi viável, analisando se houve facilidade na implementação e na estruturação do código, quais as vantagens/desvantagens e concluindo se vale a pena ou não utilizar esse tipo de

implementação para solucionar os problemas propostos. Também foram observados e descritos os padrões identificados durante a implementação dos AA com POA.

4 RESULTADOS

Nesta seção serão apresentados os resultados de todas as implementações. Inicialmente, será mostrado tudo o que foi feito em comum para as três implementações e, posteriormente, as subseções apresentarão as peculiaridades de cada exemplo.

Usualmente, a implementação de autômatos finitos é baseada em suas tabelas de transições, implementadas como matriz. Tem-se uma variável onde é armazenado o estado corrente da máquina e utiliza-se a tabela de transições para atualizá-la enquanto lê os símbolos de entrada. Isso é feito até que a palavra seja totalmente consumida.

Nesse trabalho, as implementações sem a utilização da POA foram realizadas de outra forma. Pelo fato da OO tornar a programação mais parecida com o cenário real e como a ideia era implementar as aplicações com o *AspectJ*, a implementação dos AA foi feita com o auxilio do paradigma OO e os componentes do autômato foram implementados como classes. Ou seja, foram criadas classes para representar as transições (adaptativas e não adaptativas), as funções adaptativas, os estados, os geradores e o próprio autômato.

Ao invés de implementar uma tabela de transições, cada AA possui um conjunto de de transições (objeto da classe Transição) e estados (objeto da classe Estado), implementados como um *arraylist*. Sempre que uma transição adaptativa é realizada, esse *arraylist* é modificado. Ou seja, se a transição é adaptativa, a função adaptativa correspondente é chamada e as transições do autômato são alteradas através de ações adaptativas.

As implementações com POA foram feitas de forma análoga. No entanto, algumas classes não foram necessárias. Isso porque todas as características relacionadas a adaptatividade foram implementadas no aspecto. O aspecto *AdaptacaoAspect* (Figura 12) contém o método responsável pela função adaptativa e a responsabilidade de atualizar, se existirem, os parâmetros dessa função. Assim, o aspecto implementado captura a execução da função aplicaTransicao(*) e pode inserir comportamento antes ou depois da execução dessa função.

No aspecto da Figura 12, é declarado um pointcut insereAdaptacao(Automato a, Transicao t) formado por:

1. ***Execution* (*aplicaTransicao(Transicao)):** captura a execução do método aplicaTransicao(Transicao t);

2. ***args*(t):** captura o argumento t que é passado como parâmetro pro método interceptado;

3. ***target*(a):** captura a instância que invoca o método, ou seja, o próprio autômato.

```
public aspect AdaptacaoAspect {
    //lista de parametros;

    pointcut insereAdaptacao(Automato a, Transicao t) : execution (*
        aplicaTransicao(Transicao)) && args(t) && target(a);

    after(Automato a, Transicao t): insereAdaptacao(a,t) {
        if (t!= null && t.getEstadoCorrente().getNome() == estado.getNome() &&
            t.getEstadoCorrente().getEstadofinal() == estado.getEstadofinal() &&
            t.getSimbolo() == simbolo) {
                executaFuncao(par1, par2, par3, a);
        }

    }


     public void executaFuncao(Estado i, Estado j, Estado n, Automato a) {...}

     private Transicao verificaExistenciaTransicao(Automato a, Transicao t) {...}


}
```

Figura 12- Aspecto AdaptacaoAspect

Pode-se observar também que é declarado um *advice after*, que indica que a inserção de comportamento (através da chamada da função adaptativa) será feita após a execução da função aplicaTransicao(*), presente no pointcut insereAdaptacao(Automato a, Transicao t). O método executaFuncao(*) representa a função adaptativa, que é responsável por aplicar todas as ações de remoção e inserção no AA.

O aspecto apresentado pode ser adaptado para qualquer autômato adaptativo. Para isso, deve-se:

- criar um *advice* para cada tipo de função adaptativa: função anterior (*Advice before*) e posterior(*Advice after*);

- criar um método para cada função adaptativa existente, contendo suas respectivas ações adaptativas;

- identificar o momento em que a função adaptativa será chamada. Esse momento está relacionado a uma transição do autômato.

Utilizando a POA, as implementações dos autômatos finitos ficaram totalmente independentes das adaptações. Ou seja, o modelo subjacente ficou totalmente independente. Assim, a classe Autômato poderia ser utilizada na mesma aplicação para implementar um AA e um autômato não adaptativo, o que não era possível na implementação sem POA.

4.1 IMPLEMENTAÇÃO DO SIMULADOR DE PILHA

A Figura 13 mostra como foram modeladas as classes para a implementação do simulador de pilha sem POA.

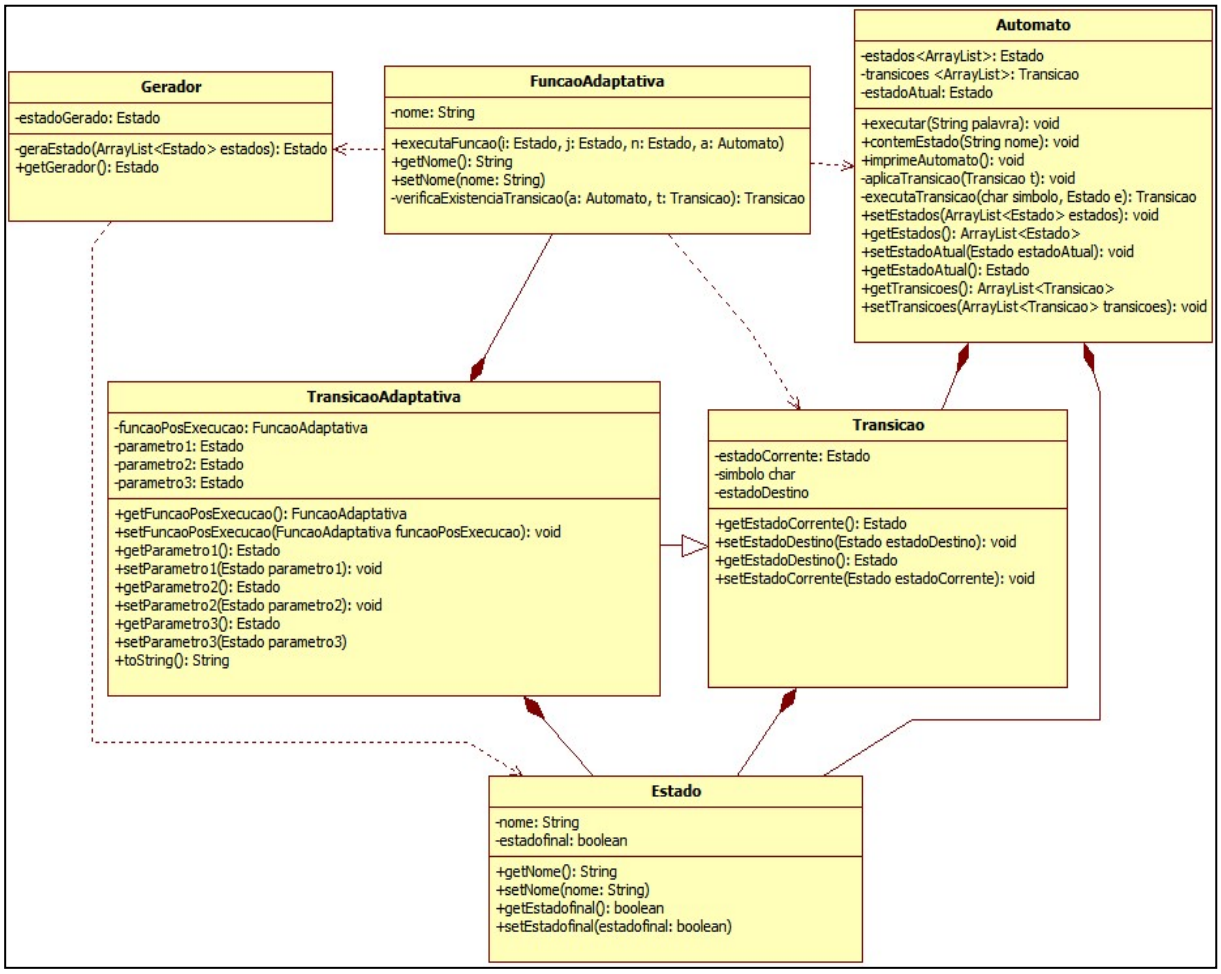

Figura 13 - Diagrama de classes do simulador de pilha sem POA

Esse exemplo possui uma única função adaptativa ***A*** que deve ser chamada logo após a execução de uma transição adaptativa. Uma característica interessante

desse modelo é que, a cada alteração do AA, a função A é chamada em um lugar diferente e com parâmetros diferentes. Ou seja, foi necessário armazenar em cada transição adaptativa criada, novos parâmetros i, j e n da função (Figura 14).

Na figura 14, é possível visualizar como as ações adaptativas da função adaptativa foram implementadas. Como o AA só possui ações de inserção e remoção, essas foram implementadas adicionando e excluindo, respectivamente, as transições do *arraylist* de transições do AA.

```
public class FuncaoAdaptativa {

    private String nome;

    public FuncaoAdaptativa(String nome) {...}

    public void executaFuncao(Estado i, Estado j, Estado n, Automato a) {
        if(a.contemEstado(i.getNome()) && a.contemEstado(j.getNome()) && a.contemEstado(n.getNome())){
            //gera o estado k
            Gerador k = new Gerador(a.getEstados());
            a.getEstados().add(k.getGerador());
            //gera o estado m
            Gerador m = new Gerador(a.getEstados());
            a.getEstados().add(m.getGerador());
            //Cria a transicao: - [ ( n, "(" ) : → i, A ( i , j , n )]
            Transicao t1 = new TransicaoAdaptativa(this, i, j, n, n, '(', i);
            if (verificaExistenciaTransicao(a, t1) != null) {
                //Remove a transicao t1, caso ela exista no automato
                a.getTransicoes().remove(verificaExistenciaTransicao(a, t1));
            }
            //Cria a transicao: + [ ( m, ")" ) : → j ]
            Transicao t2 = new Transicao(m.getGerador(), ')', j);
            if (verificaExistenciaTransicao(a, t2) == null) {
                //Adiciona a transicao t2, caso ela não exista no automato
                a.getTransicoes().add(t2);
            }
            (...)
        }

    }

    public String getNome() {...}

    public void setNome(String nome) {...}
    //Função que verifica se existe uma transição no automato a que seja igual a transição t passada por parâmetro
    private Transicao verificaExistenciaTransicao(Automato a, Transicao t) {...}
}
```

Figura 14 – Classe FuncaoAdaptativa

No simulador de pilha com POA (Figura 15), as classes que representavam a função adaptativa e a transição adaptativa não foram necessárias na implementação, conforme mencionado anteriormente. Para controlar toda a adaptatividade do AA, o aspecto passou a realizar a mudança de parâmetros da função e o local onde A é chamada. Para tal, foi preciso criar um atributo **estado** que era atualizado a cada execução da função adaptativa. Por exemplo, ***A*** é chamada depois da execução da transição entre os estados 1 e 2 na máquina E_0. Já na máquina E_1, ***A*** é chamada depois da execução da transição entre os estados 2 e 5. Dessa forma, na implementação, inicialmente o atributo **estado** recebe o estado 1, e quando o método executaFuncao(*) é chamado, ele é alterado para o estado 2. E assim sucessivamente.

Como é possível observar comparando as Figuras 13 e 15, a utilização da POA diminuiu o acoplamento entre as classes, o que facilita o entendimento e manutenção do código. Como as características da adaptação estão concentradas no aspecto, qualquer alteração dessas características será realizada somente no aspecto e não mais em diversas partes do código. Além disso, o código ficou muito mais simples, facilitando o acesso e manipulação das estruturas.

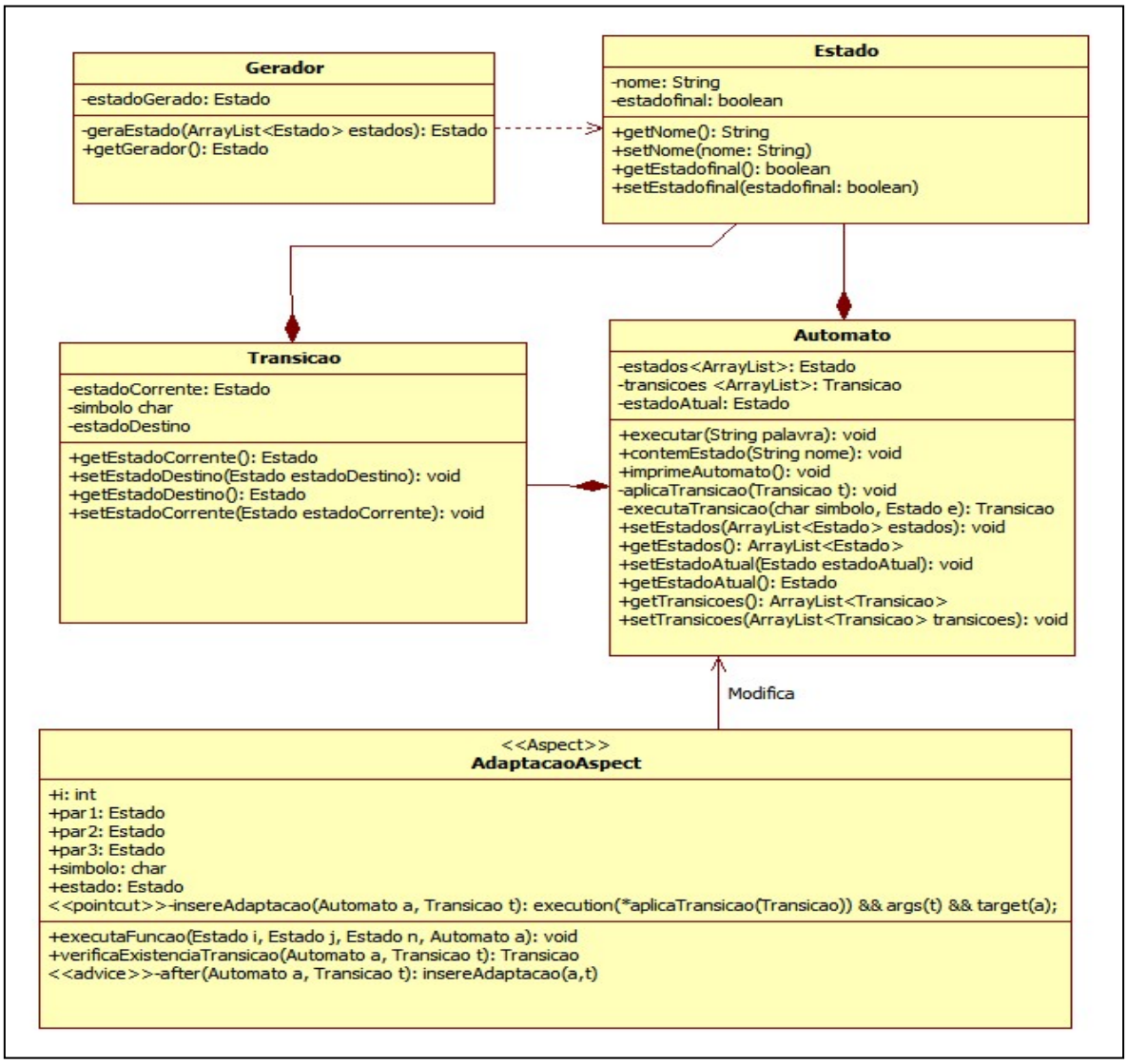

Figura 15 – Diagrama de classes do simulador de pilha com POA

4.2 IMPLEMENTAÇÃO DE DEPÊNDENCIA DE CONTEXTO

Esse AA possui duas funções adaptativas posteriores ***A*** e ***B***. O exemplo também conta com uma máquina subjacente S_2.

A implementação e a modelagem foi feita de forma análoga ao que foi apresentado na seção anterior. A diferença foi que houve a necessidade de criar uma classe APE para representar a máquina subjacente e a classe Autômato passou a ter um atributo desse tipo. Utilizou-se um APE como máquina subjacente, pois a máquina S_2 fazia chamadas recursivas a ela mesma e ao fim de cada chamada era preciso que a execução da máquina retornasse a um determinado estado de S_2. Por isso, foi utilizado um APE, já que esse modelo possui uma pilha com esse objetivo.

A classe APE tem a mesma estrutura da classe Automato já apresentada (Figuras 13 e 15), porém foi adicionado um *arraylist* de estados para representar a pilha.

Outra peculiaridade dessa aplicação foi a exclusão da classe Gerador, uma vez que não são criados novos estados. As funções adaptativas apenas inserem novas transições entre os estados já existentes.

Na versão com POA, foram criados dois métodos, um para cada função adaptativa e o aspecto foi basicamente igual ao apresentado para o exemplo Simulador de Pilha. Nesse caso, as funções ***A*** e ***B*** eram chamadas sempre que na máquina PROG era lido um símbolo “a” e/ou “b”, respectivamente. Ou seja, o símbolo lido foi a forma de identificar onde as funções seriam chamadas no aspecto.

As vantagens observadas foram as mesmas do simulador de pilha.

4.3 IMPLEMENTAÇÃO DE RECUPERAÇÃO DE ERROS SIMPLES

Esse AA tem a característica de invocar a função adaptativa sempre que ocorre um erro simples), isto é, a função não é chamada em um lugar específico. Por isso, a implementação dispensou o uso da classe TransicaoAdaptativa, pois a função adaptativa não está ligada a nenhum tipo de transição. Então, todas as transições do autômato são transições comuns de AF.

A implementação foi realizada de forma que o AA fosse capaz de se recuperar de erros simples em qualquer linguagem. Para isso, basta definir a linguagem e as transições e inserir no AA. Foi implementado o algoritmo de recuperação de erros apresentado na Figura 3.

Em virtude da função θ utilizada no algoritmo, que denota o conjunto $\Sigma - firstSymbols(q)$, foi adicionado um *arraylist* de caracteres a classe Automato, representando o alfabeto da linguagem. O restante da implementação foi análoga as anteriores.

Na versão com POA, a diferença é a chamada da função adaptativa, que ocorre sempre que a função interceptada (aplicaTransicao(*)) recebe uma transição nula por parâmetro. Isso representa que, para o estado corrente e símbolo lido não há qualquer transição na lista de transições do AA, o que caracteriza um erro. Vale ressaltar que, em um AF sempre há transição para todos os símbolos lidos, mesmo que seja para um estado de erro. Isso não foi considerado nesse trabalho. Não foram criadas transições de erro.

Resumindo, foram observados alguns padrões de desenvolvimento nas implementações realizadas como pode ser visto a seguir. A partir desses padrões, é possível fazer um mapeamento entre os componentes de um AA e como eles podem ser implementados utilizando POA. Esse mapeamento é apresentado no Quadro 2:

Componente do AA	**Recurso na POA**
Função adaptativa	- *Advice* para chamar a função - Método que implementa as ações adaptativas da função - (Opcional) Parâmetros e/ou métodos auxiliares
Função adaptativa anterior	*Advice before*
Função adaptativa posterior	*Advice after*
joinpoint	Execução do método aplicaTransicao(*)
pointcut	Métodos *args*(t), *target*(a) e execução do método aplicaTransicao(*)

Quadro 2 – Mapeamento do AA para Aspecto

- Cada função adaptativa deve ser chamada a partir de um *advice* no aspecto, já que representa o comportamento a ser inserido em um determinado ponto. Cada *advice* é responsável por verificar se as transições interceptadas através dos pointcuts são adaptativas e, se forem, invocar a função adaptativa correspondente;
- Deverá existir um método para cada função adaptativa do modelo contendo suas respectivas ações adaptativas;

- Se as funções adaptativas possuem parâmetros, esses deverão ser mantidos e atualizados dentro do aspecto. Dependendo da quantidade de funções adaptativas e parâmetros no aspecto, isso pode gerar uma confusão no código, deixando-o muito sobrecarregado e diminuindo sua legibilidade;
- Funções adaptativas anteriores entram em ação antes de um *joinpoint*. Assim, o *advice* deve ser do tipo *before*.
- Funções adaptativas posteriores entram em ação depois de um *joinpoint*. Assim, o *advice* deve ser do tipo *after*.
- O *joinpoint* será sempre a execução do método aplicaTransicao(*);
- É necessário criar um *pointcut* no aspecto para declarar o *joinpoint* da execução da função aplicaTransicao(*).
- O *pointcut* será composto também pelos métodos *args*(t) e *target*(a) para que ele consiga ter acesso a transição passada por parâmetro no método aplicaTransicao(*) e a instância do AA;

A aplicação pode ter vários aspectos, desde que sejam utilizados de forma análoga ao que foi apresentado. Pode existir um aspecto para cada função adaptativa por exemplo.

5 CONCLUSÃO

Nesse trabalho foi realizada a implementação de três autômatos adaptativos com e sem a utilização de POA.

As implementações realizadas comprovam a viabilidade da implementação de AA com POA. É importante ressaltar que a solução apresenta uma forma de implementação, outras podem ser realizadas.

A partir das implementações dos AA escolhidos, vários padrões de desenvolvimento foram observados e descritos, como pode ser visto no final da sessão de resultados, o que pode auxiliar futuros desenvolvedores na implementação de AA com POA.

Várias vantagens foram observadas na implementação com POA. As principais foram: a diminuição de acoplamento, facilidade de manipulação das estruturas, boa modularização, aumento da legibilidade e confiabilidade, organização do código e, consequentemente, facilidade de manutenção do mesmo. Além disso, observou-se a possibilidade de criar os dois tipos de autômato, adaptativo e não adaptativo, na mesma aplicação, utilizando a mesma classe. A principal desvantagem observada é que o aspecto pode ficar sobrecarregado e causar confusão no código em sistemas que demandem muitas funções adaptativas.

Os objetivos do trabalho foram alcançados e acredita-se que vale a pena combinar AA com POA pelas inúmeras características e benefícios apresentados.

Como trabalho futuro, pretende-se desenvolver um framework para implementação de AA com POA. A ideia surgiu a partir da observação de que boa parte do código foi a mesma para as três implementações. No framework será fornecida a estrutura geral de implementação e o usuário poderá inserir as peculiaridades dos seus modelos de acordo com as suas necessidades.

6 REFERÊNCIAS

BARROS, M. B. *Uma Abordagem Genérica Para Gridificação De Aplicações Concorrentes Utilizando Programação Orientada A Aspectos*. 2010. 91 f. Dissertação (Mestrado) - Curso de Informática Aplicada, Universidade de Fortaleza, Fortaleza, 2010. Cap. 3.

CAMPOS, J. P. *Extração De Candidatos A Aspectos A Partir De Descrições De Fluxo De Casos De Uso*. 2009. 102 f. Dissertação (Mestrado) - Curso de Pós Graduação em Ciência da Computação, Universidade Federal de Viçosa, Viçosa, 2009.

CASACHI, R.; CAMOLESI, A. R. Uso De Programação Orientada A Aspecto No Desenvolvimento De Aplicações Que Utilizam Conceitos De Tecnologia Adaptativa. *Revista de Sistemas e Computação-RSC*, v. 2, n. 1, 2012.

JESUS, L. D.; SANTOS, D. G. D.; CASTRO, A. A. D.; PISTORI, H. WTA 2007-II. 2-AdapTools 2.0: Implementation And Utilization Aspects. Latin America Transactions, *Revista IEEE América Latina*, v. 5, n. 7, p. 527-532, 2007.

KITCHENHAM, B. Procedures For Performing Systematic Reviews. Keele, UK, *Keele University*, v. 33, p. 2004, 2004.

LADDAD, R. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press, 2003.

MENEZES, P. B. *Linguagens Formais E Autômatos*. 6. ed., Porto Alegre: Bookman, 2011.

NETO, J. J. *Contribuições À Metodologia De Construção De Compiladores*. São Paulo: Tese de Livre Docência, USP, 1993.

NETO, J. J. Solving Complex Problems Efficiently With Adaptive Automata. In: Implementation And Application Of Automata. *Springer Berlin Heidelberg*, 2001. p. 340-342.

NETO, J. J. Adaptive Rule-Driven Devices-General Formulation And Case Study. In: Implementation And Application Of Automata. *Springer Berlin Heidelberg*, 2002. p. 234-250.

NETO, J. J. Autômatos Em Engenharia De Computação – Uma Visão Unificada. *Primera Semana de Ciencia y Tecnología de la Sociedad Chotana de Ciencias y la Red Mundial de Científicos Peruanos*, p. 22-27, 2003.

NETO, J. J. Um Levantamento Da Evolução Da Adaptatividade E Da Tecnologia Adaptativa. *Revista IEEE América Latina*, v. 5, n. 7, p. 1548-0992, 2007.

NETO, J. J. Um Levantamento Da Pesquisa Em Técnicas Adaptativas Na Epusp//A Research Survey On Adaptive Techniques At The University Of Sao Paulo, Brazil. *Revista de Sistemas e Computação-RSC*, v. 1, n. 1, 2011.

NETO, J. J.; ROCHA, R. L. A. *Autômatos Adaptativos*. Disponível em: <http://www.pcs.usp.br/~lta/disciplinas/fun_apl_tec_ada/aula04_automatos_adaptativos.pdf>. Acesso em: 28 jan. 2014.

OSSHER, H.; TARR, P**.** Using Multidimensional Separation Of Concerns To (Re)Shape Evolving Software**.** *Communications of the ACM*, New York, v. 44, n. 10, p 43-50, Oct. 2001.

PISTORI, H. *Tecnologia Adaptativa Em Engenharia De Computação: Estado Da Arte E Aplicações.* Tese de Doutorado, Escola Politécnica da USP, São Paulo, 2003.

PIVETA, E. K. Aurélia: *Um Modelo De Suporte A Programação Orientada A Aspectos***.** 2001. 80 f. Dissertação (Mestrado) - Curso de Mestre em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis, 2001.

RAMOS, M. V. M.; NETO, J. J.; VEGA, Í. S. *Linguagens Formais: Teoria, Modelagem E Implementação*. Porto Alegre: Bookman, 2009.

RESENDE, A. M. P.; SILVA, C. C. Programação Orientada A Aspectos Em Java. *Rio de Janeiro: Brasport*, 2005. 176 p.

SILVEIRA, K. K. *Uma Estratégia Baseada Em Programação Orientada A Aspectos Para Injeção De Falhas De Comunicação***.** 2005. 75 f. Dissertação (Mestrado) -

Curso de Pós Graduação em Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

SOARES, A. H. V.; ROCHA, A. R.; ALVES, F. L.; ALVES, J. C. *Programação Orientada A Aspectos: Uma Visão Geral*. Disponível em: <http://www.ic.unicamp.br/~rocha/college/src/aop.pdf>. Acesso em: 28 jan. 2014.

TANIWAKI, C. Y. O.; NETO, J. J. *Autômatos Adaptativos No Tratamento Sintático De Linguagem Natural.* Technical report, PCS-POLI-USP, São Paulo, Brasil, 2001.

TIRELO, F.; BIGONHA, R. S.; BIGONHA, M. A. S.; VALENTE, M. T. O. *Desenvolvimento De Software Orientado Por Aspectos***.** 2004. Disponível em: <http://homepages.dcc.ufmg.br/~mtov/pub/jai2004.pdf>. Acesso em: 28 jan. 2014.

VIEIRA, N. J. *Introdução aos fundamentos da computação: linguagens e máquinas*. Pioneira Thomson Learning, 2006.

ZUFFO, F. A.; PISTORI, H. Tecnologia Adaptativa E Síntese De Voz: Primeiros Experimentos. *Prefácio 5º WSL*, p. 165, 2004.