Especificações adaptativas e objetos: Uma técnica de *design* de software a partir de *statecharts* com métodos adaptativos

Ítalo S. Vega

***Abstract*— The use of flowcharts for the modularization of software applications is a common practice in structured development. However, there is a recommendation to avoid the use of this tool for decomposing an application into modules. This paper presents a proposal to modularization in terms of objects from adaptive specifications. Over the course of designing the software architecture, various artifacts are produced, with emphasis on the state transition tables extended with adaptive methods. The resulting architecture defines static dependencies that facilitate the understanding of the implementation model and management of a software development project.**

Keywords—adaptive specifications, adaptive methods, static dependencies, software architecture, design patterns, principles of object modeling.

I. Introdução

O DESENVOLVIMENTO de aplicações de software deve ser conduzido de tal forma que o funcionamento de um computador automático programado ou, simplesmente, computador, atenda às necessidades e metas (NMs) do seu usuário. Nesse sentido, a disciplina de análise de um processo de desenvolvimento de software [1], [2] assume o propósito fundamental de especificar os requisitos de tal funcionamento:quais condições ou capacidades o computador deverá exibir para que o usuário seja atendido nas suas NMs [3]. Atuando como engenheiro de software, um programador assumirá a responsabilidade de elaborar um programa contendo instruções executáveis pelo computador e que o façam operar conforme a especificação declarada de requisitos.

A. Organização de instruções

Mas como elaborar tal programa? Uma forma é modelar o funcionamento do computador (percebido como uma máquina de estados) por descrição dos passos computacionais a serem por ele realizados. Este modelo da computação (sequência de passos computacionais) torna-se a base para que o programador elabore as instruções do programa, o que é feito pelo emprego de técnicas de *design* (projeto) e de implementação [2]. Neste sentido, Parnas descreve duas estratégias para a programação de um computador por modularização de instruções: (i) decomposição funcional e (ii) decomposição por tipos de dados abstratos. Muito embora as instruções estejam diretamente relacionadas aos passos computacionais, também torna-se importante, durante, a programação, a particular maneira como tais instruções são organizadas na estrutura do programa. Parnas [4], p. 1058, afirma que não se deve organizar as instruções utilizando a estratégia de decomposição funcional considerando apenas a sequência dos passos computacionais ao longo do tempo[1]:

> "We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart ... Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing."

Mas, se a recomendação é não derivar a organização das instruções de um programa a partir de um fluxograma, que alternativas devem ser consideradas? Seguindo a linha de Parnas, as instruções devem ser organizadas por tipos de dados abstratos cujas execuções são coordenadas por um módulo de controle [4], p. 1054.

B. Máquinas de estados

Uma outra possibilidade é expressar o funcionamento desejado por meio de uma máquina de estados que modela o comportamento do computador. Douglass [5] define uma máquina de estados da seguinte maneira[2]:

> "A finite state machine is a machine specified by a finite set of conditions of existence (called states) and a likewise finite set of transitions among states triggered by events".

Este artigo assume tal definição de máquina de estados, de modo que os comportamentos de um computador sejam essen-

Ítalo S. Vega trabalha no Departamento de Computação da Faculdade de Ciências Exatas e Tecnologia da Pontifícia Universidade Católica de São Paulo, São Paulo, capital, Brasil e-mail: italo@pucsp.br.

[1] Tradução livre: "Tentamos demonstrar, por esses exemplos, que é quase sempre incorreto começar a decomposição de um sistema em módulos com base em um fluxograma ... Assim, como na maioria dos casos as decisões de projeto transcendem o tempo de execução, módulos não corresponderão a passos do processamento."

[2] Tradução livre: "Uma máquina de estados finita é uma máquina especificada por um conjunto finito de condições de existência (chamados estados) e um igualmente conjunto finito de transições entre os estados, disparadas por eventos."

cialmente descritos, em modelos, por meio de estados e transições. É comum a representação de máquinas de estados por meio de tabelas de transição. Elas suportam a exploração sistemática do modelo de uma máquina que exibe comportamentos dependentes da sua história. Outra representação de máquinas de estados é proposta por Harel. Trata-se de um formalismo visual para representar modelos de máquinas cujo comportamento pode ser expresso por um conjunto de estados discretos conhecido por diagramas (de transição) de estados. O formalismo oferece suporte à especificação hierárquica de estados e à evolução concorrente de eventos [6].

A partir de uma especificação comportamental formulada por uma máquina de estados, diversas estratégias de implementação podem ser aplicadas para a obtenção das instruções e da sua organização:

- sequências de instruções de decisão a partir de uma condição expressa pelo valor de uma variável de estado [7];
- uma estrutura de dados matricial utilizada por uma lógica de controle [8].

Entretanto, tais estratégias conduzem a uma organização na qual as instruções encontram-se fortemente acopladas [1], [9]. A consequência primária deste acoplamento é que eventuais edições de instruções relacionadas com uma parte do modelo comportamental desencadeiam reedições (ao menos recompilações) de outras instruções que se encontram em módulos distintos. Dificuldades decorrentes de alterações em módulos que não se referem à implementação da funcionalidade revisada são resultado de uma arquitetura ruim [10], [11].

O paradigma de orientação a objetos introduz técnicas mais apropriadas para lidar com o problema de organização das instruções de um programa.

C. Paradigma de objetos

Na visão de Rumbaugh et al., um diagrama de estados deve ser utilizado para descrever os comportamentos de uma única classe de objetos [12], p. 90. Ou seja, uma classe, no paradigma de objetos, atua como elemento de organização de comportamentos no modelo de implementação. Por conseguinte, o programa será constituído por um conjunto de classes de objetos cujos comportamentos podem ser descritos por diagramas de estados.

Agora, seguindo Parnas, as classes não devem ser originárias dos passos computacionais que especificam o funcionamento de um computador. À medida que o funcionamento da máquina passa a ser descrito por um comportamento mais complexo, é de se esperar que sejam necessárias cada vez mais instruções para a realização dos passos computacionais. Lakos investiga a influência das dependências físicas (dependências entre instruções) em projetos de software de larga escala [11]. Ele reforça a importância da modularização e do desacoplamento entre grupos de instruções. Garlan e Shaw, por sua vez, apresentam diversos estilos arquiteturais para a organização das instruções de um programa. Um deles é o orientado a objetos [9]. Isso leva ao uso do paradigma de objetos como um modelo com foco primário na organização das instruções que constituem um programa de computador.

D. Programas e máquinas

Jackson também se preocupa com a estrutura de sistemas de software de grande porte [13]. Mas ele investiga uma outra abordagem para o problema. De acordo com Jackson, um programa é uma "descrição de uma máquina" [14]. Neste sentido, ele propõe que a máquina deve ser construída por programação; ela resulta da execução das instruções por um computador de propósito geral [15].

A Fig. 1 ilustra a ideia de Jackson, enfatizando que uma máquina concreta, ao executar as instruções de um programa, origina uma nova máquina em tempo de execução. Sob a perspectiva funcional, um programador deve elaborar um programa que descreva as funcionalidades da máquina concreta. Sob a perspectiva de Jackson, no entanto, um programador deve se preocupar com a descrição de uma nova máquina — programas como descrições de máquinas.

Figura 1. Programas e máquinas

E. Organização do artigo

Portanto pode-se elaborar um programa considerando que ele irá descrever uma nova máquina (virtual) cujo comportamento pode ser especificado por diagramas de estados. Este artigo apresenta uma proposta para a elaboração de programas que descrevem máquinas partindo de especificações comportamentais adaptativas. Na Seção II, aplicam-se as técnicas tabelas-verdade, tabelas de transição e tabelas de decisão na especificação funcional e comportamental de uma aplicação de software. Na Seção III, estendem-se as tabelas de decisão com métodos adaptativos, visando o suporte à reconfiguração de estados de uma máquina. Na Seção IV, emprega-se um padrão de projeto como base para originar uma arquitetura organizada por estados, mas com propriedades ruins de dependências Na Seção V, aplica-se um princípio de modelagem para revisar a arquitetura, com preservação de funcionalidade. Finalmente, na Seção VI conduz-se uma nova

revisão arquietural com suporte a edições sistemáticas das instruções do modelo de implementação.

II. Requisitos Funcionais e Comportamentais

O problema de avaliação de um aluno ao longo de um período letivo servirá como contexto de discussão neste trabalho. A avaliação envolverá a realização de até três provas, na seguinte ordem de ocorrência: *P1*, *P2* e *PS*. É em função das notas de tais provas que se calculará a média final do aluno para determinar a sua situação de aprovação.

A. Requisitos Funcionais

Um requisito funcional pode ser entendido como uma espécie de promessa de funcionamento da máquina. Parte do problema de programação se refere ao projeto de uma máquina cujo comportamento atenda à declarada promessa de funcionamento. A técnica de tabelas-verdade ajuda na identificação das condições que devem ser verdadeiras para acionar uma funcionalidade da máquina.

1) Condições de cálculo: As condições de cálculo da média final de um aluno serão expressas na forma de uma tabela-verdade. A Tab. I especifica sob quais condições as funcionalidades de cálculo poderão ser solicitadas. Por exemplo, na condição 1 (primeira linha da tabela), é verdade que o aluno realizou (fez) as três provas e que o período letivo foi encerrado. É possível, nesta condição, aplicar a fórmula de cálculo da média do aluno e determinar se foi ou não aprovado. Nesta situação, diz-se que o aluno foi avaliado. Um par de colchetes será utilizado para indicar uma condição. Um aluno foi avaliado se a condição [Avaliado] for verdadeira.

TABELA I

CONDIÇÕES DE AVALIAÇÃO

	[fpl]	[fezP1]	[fezP2]	[fezPS]	[Avaliado]
1	T	T	T	T	T
2	T	T	T	F	T
3	T	T	F	T	T
4	T	T	F	F	T
5	T	F	T	T	T
6	T	F	T	F	T
7	T	F	F	T	F
8	T	F	F	F	T
9	F	—	—	—	F

As condições 7, 9 e seguintes referem-se àquelas nas quais as funcionalidades de cálculo para a avaliação do aluno no período letivo não podem ser acionadas. Na condição 7, não se deve considerar o caso de alunos que fizeram apenas a prova PS, por se tratar de uma situação que não pode ocorrer neste domínio. Nos outros casos, não se pode calcular a situação do aluno sem o término do período letivo.

2) Ações de cálculo: As ações de cálculo nomeiam as funcionalidades da máquina em desenvolvimento. Elas serão apresentadas na forma de uma tabela de decisão, conforme descrito por Rowlett [16]. A Tab. II especifica os possíveis cenários de avaliação de um aluno. As ações (funcionalidades) e a ordem na qual deverão ser realizadas para o correto cálculo da situação no aluno são indicadas para cada um dos cenários de funcionamento.

No cenário 1, é verdade que o aluno realizou as três avaliações e houve encerramento do período letivo. As seguintes ações deverão ser realizadas. Primeiro, será feito o cálculo da sua média final pela aplicação da regra associada à ação p1p2psmf. Isto resultará no valor *MF*. Em seguida, aplicarse-á a regra da ação situaçãoMF, que resultará no valor *SIT*, considerando *MF*. Este valor indica aprovação ou reprovação do aluno. É verdade que um aluno estará [Avaliado] quando ele souber o valor de *MF* e o valor de *SIT* do seu caso.

TABELA II

FUNCIONALIDADES PARA A AVALIAÇÃO DE UM ALUNO

	Cenário						
	1	2	3	4	5	6	7
Condição							
[fpl]	T	T	T	T	T	T	T
[fezP1]	T	T	T	T	F	F	F
[fezP2]	T	T	F	F	T	T	F
[fezPS]	T	F	T	F	T	F	F
Ação							
p1p2psmf	1						
p1p2mf		1					
p1psmf			1				
p2psmf					1		
situaçãoMF	2	2	2		2		
situação				1		1	1

3) Fórmulas de cálculo: As especificações das fórmulas associadas ao cálculo da média final de um aluno, correspondente ao valor *MF*, encontram-se na Tab. III. A fórmula da ação p1p2psmf faz uso da função max(lista). Ao ser avaliada, tal função retornará a soma das duas maiores notas fornecidas no argumento lista. As demais fórmulas são de interpretação imediata.

TABELA III

FÓRMULAS PARA O CÁLCULO DA MÉDIA FINAL

Ação	Fórmula de *MF*
p1p2psmf	$\frac{\max(P1,P2,PS)}{2}$
p1p2mf	$\frac{P1+P2}{2}$
p1psmf	$\frac{P1+PS}{2}$
p2psmf	$\frac{P2+PS}{2}$

As regras que se referem ao cálculo da situação de um aluno encontram-se especificadas na Tab. IV. Estas regras encontram-se associadas às ações que determinam a situação de um aluno no final do período letivo, denotado por *SIT*. Na ação situaçãoMF, a aplicação da fórmula depende do valor *MF* calculado em um passo anterior. A fórmula situação, por outro lado, retorna um valor constante *RP*.

TABELA IV

FÓRMULAS PARA O CÁLCULO DA SITUAÇÃO

Ação	Fórmula de *SIT*
situaçãoMF	*AP*, se $MF \geq 5$
	RP, se $MF < 5$
situação	*RP*

A ordem na qual as ações deverão ocorrer ao longo do tempo é parte da especificação comportamental, tema da Subseção II-B.

B. ESPECIFICAÇÃO COMPORTAMENTAL

Uma categoria de modelos para lidar com problemas de sequenciamento de eventos no tempo baseia-se em autômatos de estados finitos [6], [17], [18]. Tais modelos podem ser utilizados para especificar particulares sequências de eventos considerando-se o estado corrente e o histórico de ocorrências.

Autômatos de estados finitos podem ser representados por tabelas de transição de estados. Em tais tabelas, cada coluna denota um tipo de evento e cada linha se refere a um estado do autômato. O preenchimento da tabela é feito de tal forma a indicar o estado seguinte quando da ocorrência de um evento do tipo nomeado pela respectiva coluna. Utiliza-se o símbolo — para indicar que o evento não provoca transição de estado e o símbolo → para mostrar qual o estado inicial do autômato.

TABELA V

TABELA DE TRANSIÇÕES DE ESTADOS DO PROBLEMA

	Estado	Evento			
		fezP1	fezP2	fezPS	fpl
→	**NãoAvaliado**	**2**	**3**	—	**4**
2	**FaltaP2PS**	—	**5**	**6**	**4**
3	**FaltaPS**	—	—	**7**	**4**
4	**Avaliado**	—	—	—	—
5	**P1P2MF**	—	—	**8**	**4**
6	**P1PSMF**	—	—	—	**4**
7	**P2PSMF**	—	—	—	**4**
8	**P1P2PSMF**	—	—	—	**4**

A especificação das sequências de eventos para o caso em estudo encontra-se na Tab. V. Partindo-se do estado inicial de um aluno que ainda não realizou provas no período letivo, indicado por **NãoAvaliado**, três situações podem ocorrer: realização da prova *P1* ou da prova *P2* ou o encerramento do período letivo sem a realização de provas. Ou seja, partindo-se do estado **NãoAvaliado**, apenas três transições originadas neste estado deverão ser permitidas. No texto, tipos de eventos serão indicados por uma sublinha ondulada. Assim, partindo-se do estado **NãoAvaliado** da Tab. V, na ocorrência de um evento do tipo

- fezP1, o autômato passa para o estado **FaltaP2PS** (representado por **2**);
- fezP2, autômato passa para o estado **FaltaPS** (representado por **3**);
- fpl, autômato passa para o estado Avaliado (representado por **4**).

1) Visualização por diagramas de transição de estados: O modelo comportamental pode ainda ser descrito por um Diagrama de Máquina de Estados UML (DME) [19] com a semântica estabelecida por Harel. A Fig. 2 mostra o diagrama deste modelo cujo propósito é especificar a semântica da ordem de ocorrência das notas das provas realizadas ao longo do tempo. No diagrama, o estado inicial é **NãoAvaliado**. Três transições partem deste estado, nomeadas de acordo com o tipo do evento de disparo: fezP1, fezP2 e fpl. Na ocorrência de um evento do tipo fezP1, passa-se para o estado **FaltaP2PS** e assim por diante.

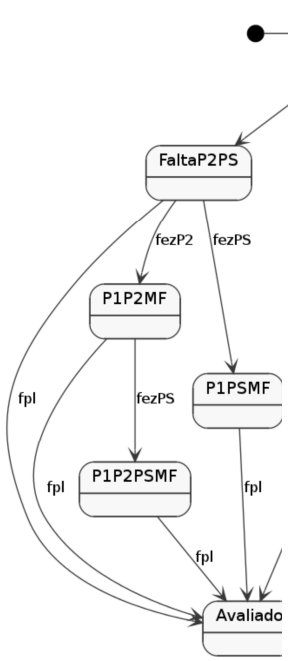

Figura 3. Vinculação de fórmulas ao modelo de estados (DME)

2) Ações de cálculo: Inspirado na notação de Máquinas de Mealy [18], DMEs podem ser anotados com ações semânticas. O diagrama da Fig. 3 ilustra uma parte do modelo comportamental com ênfase na vinculação das fórmulas de cálculo do valor *MF*, considerando-se que, inicialmente, a prova *P1* foi realizada.

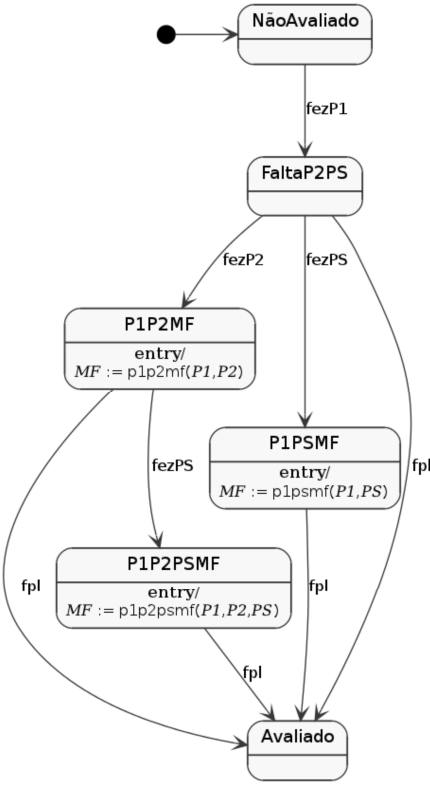

Figura 2. Especificação comportamental do estudo de caso (DME)

No estado **FaltaP2PS**, o passo computacional *MF*:=p1p2mf(*P1*,*P2*) será realizado quando da ocorrência de um evento do tipo fezP2. A sua realização calculará a média final do aluno a partir das notas *P1* e *P2*. Este comportamento ocorrerá na entrada do estado **P1P2MF**. Tal tipo especial de evento é indicado por entry.

Deve-se interpretar os demais casos envolvendo as ações associadas aos eventos do tipo entry desta mesma forma. Neste diagrama, observa-se que a organização das ações de cálculo do valor *MF* é determinada pelos diferentes estados da máquina. Na transição para o estado **P1P2MF**, aplica-se a fórmula p1p2mf e assim por diante.

III. Especificações Adaptativas

Na Seção I foram apresentadas diversas estratégias para a obtenção e organização das instruções que compõem um programa. Neste trabalho, entretanto, será proposta uma estratégia alternativa para a elaboração do modelo de implementação. Especificações utilizando autômatos adaptativos dirigidos por regras serão consideradas como ponto de partida para a elaboração de um modelo de implementação baseado no paradigma de objetos.

TABELA VI

Tabela de transições adaptativas de estados do problema

	Estado	Evento			
		fezP1	fezP2	fezPS	fpl
→	**NãoAvaliado**	ma1P1	ma1P2	—	fa1fpl
		2	**3**	—	**4**
2	**FaltaP2PS**	—	ma2P2	ma2PS	ma2fpl
		—	**5**	**6**	**4**
3	**FaltaPS**	—	—	ma3PS	ma3fpl
		—	—	**7**	**4**
4	**Avaliado**	—	—	—	—
		—	—	—	—
5	**P1P2MF**	—	—	ma5PS	ma5fpl
		—	—	**8**	**4**
6	**P1PSMF**	—	—	—	ma6fpl
		—	—	—	**4**
7	**P2PSMF**	—	—	—	ma7fpl
		—	—	—	**4**
8	**P1P2PSMF**	—	—	—	ma8fpl
		—	—	—	**4**

Um autômato adaptativo é um autômato que possui a capacidade de alterar a sua própria estrutura de estados e transições (topologia) por meio da realização de ações adaptativas de inserção e de remoção, conforme definido por Neto [20]. No caso de uma representação na forma de diagrama de transições de estado adaptativas, Vega propõe uma semântica operacional implementada por um modelo de objetos [21]. O trabalho de Neto et al. [22] a respeito de *statecharts adaptativos*, servirá como base para a elaboração de um modelo comportamental para o caso da avaliação de um aluno ao longo de um período letivo.

A. *Especificações comportamentais com funções adaptativas*

Para especificar a ordem de ocorrência de tipos de eventos com autômatos adaptativos, será feito uso de uma abstração da semântica das funções adaptativas proposta por Neto [20] denominada métodos adaptativos.

1) Métodos Adaptativos: A cada transição do autômato finito subjacente, será vinculada uma abstração do efeito conjunto das funções adaptativas *before* e *after* sobre a configuração da topologia de estados e transições do autômato. Deste modo, na ocorrência de um evento, altera-se o escopo da semântica de transição do autômato subjacente de acordo com o resultado da execução do método adaptativo a ela associada. Isso poderá provocar uma reconfiguração na topologia do autômato subjacente, em termos similares aos quais um autômato adaptativo se altera por força da aplicação de alguma função adaptativa.

Uma apropriada combinação de tais efeitos dos métodos adaptativos deverá ser o objetivo durante a elaboração de uma especificação de requisitos comportamentais. Associada a cada transição, dever-se-á descrever quais mudanças a configuração do autômato deverá sofrer para que se expresse o comportamento desejado.

2) Especificação Adaptativa: A Tab. VI apresenta o resultado da vinculação de abstrações de funções adaptativas (métodos adaptativos) no estudo de caso deste trabalho.

Na Fig. 4 encontra-se um diagrama de transições de estado estendido para expressar os métodos adaptativos (DMA) da configuração inicial da topologia do autômato adaptativo que modela a especificação comportamental do estudo em andamento. Os nomes das transições correspondem aos nomes sublinhados dos tipos de eventos vinculados. Precedendo o nome do tipo do evento, encontra-se o nome do método adaptativo associado à transição.

Uma vez nomeados e vinculados os métodos adaptativos a cada uma das transições, parte-se para a modelagem das alterações topológicas que caracterizam o comportamento da máquina sendo projetada.

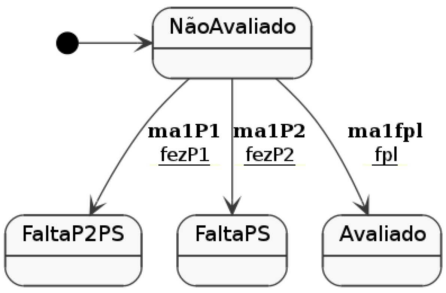

Figura 4. Configuração inicial da máquina de estados adaptativa (DMA)

3) Efeito de ma1P1 : O efeito do método adaptativo ma1P1 deve ser tal que a configuração da topologia do autômato seja alterada para aquela apresentada na Fig. 5. Ocorrendo um evento do tipo que dispare a transição fezP1 no modelo da configuração ilustrada na Fig. 4, a topologia deverá ser alterada de acordo com o modelo representado pelo diagrama da Fig. 5. Observe-se que o estado **FaltaP2PS** tornou-se inicial na nova configuração da topologia do autômato.

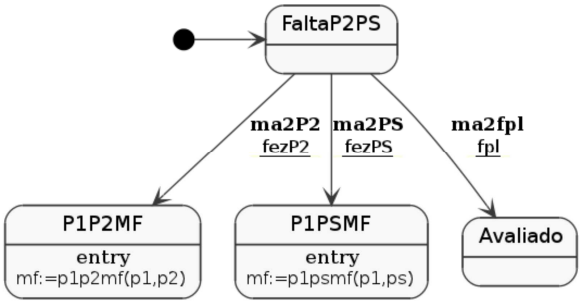

Figura 5. Efeito do método adaptativo ma1P1 (DMA)

4) Efeito de ma1P2 : A ocorrência de eventos que disparem transições do tipo fezP2 durante o estado inicial **NãoAvaliado** deverá provocar uma nova transformação na configuração da topologia do autômato (Fig. 6). A execução do método adaptativo ma1P2 tornará inicial o estado **FaltaPS** e novas transições com métodos adaptativos farão parte da reconfiguração topológica. Apenas dois outros estados podem ser atingidos nesta nova configuração.

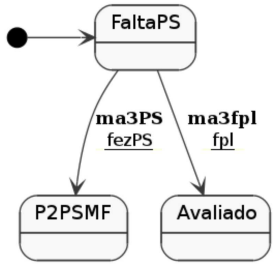

Figura 6. Efeito do método adaptativo ma1P2 (DMA)

5) Efeito de ma1fpl : A ocorrência de eventos que disparam a transição fpl na configuração ilustrada no diagrama da Fig. 4 alterará a configuração da topologia conforme o modelo da Fig. 7. Nesta configuração existe apenas o estado inicial **Avaliado**: houve encerramento do período letivo e o aluno não realizou prova alguma.

Figura 7. Efeito do método adaptativo ma1fpl (DMA)

6) Efeitos dos demais métodos adaptativos: A diagramação dos modelos referentes às demais mudanças de configurações segue uma linha de raciocínio similar. A título de ilustração, o diagrama da Fig. 8 ilustra a configuração de estados que resulta da execução dos métodos adaptativos referentes à sequência de transições fezP1, fezP2 e aluno avaliado devido ao encerramento do período letivo (fpl).

Isto conclui a elaboração da especificação adaptativa do estudo de caso. Com base em tal especificação, dever-se-á elaborar o modelo de uma máquina que se comporte conforme tal especificação. O projeto do modelo seguirá o paradigma de objetos para a concepção da arquitetura de implementação.

IV. ORGANIZAÇÃO COM *STATE*

Um possível modelo orientado a objetos para a implementação de máquinas de estados segue a proposta do padrão de projeto *State* de Gamma et al. [23]. A Fig. 9 ilustra a estrutura do padrão por meio de um diagrama de classes UML (DCL). A variabilidade comportamental da classe-papel Contexto é encapsulada na hierarquia de classes que se origina no ancestral cujo tipo-papel é *Estado*. Este tipo define uma interface de operações que devem ser implementadas de forma apropriada, em função do estado que se encontram os objetos da classe-papel Contexto. Cada descendente implementa a operação que irá determinar o comportamento do Contexto em um dos seus particulares estados.

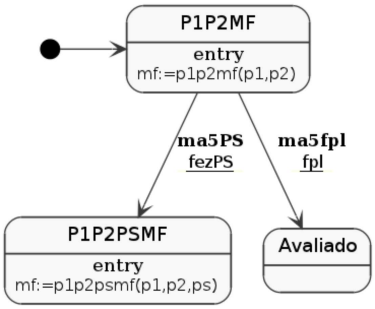

Figura 8. Efeito do método adaptativo ma1P2 (DMA)

No estudo de caso, a topologia adaptativa do autômato assume o papel de Contexto e as diferentes configurações topológicas serão vistas como sendo do tipo *Estado*, ancestral da hierarquia de classes do padrão *State*. Desta forma, tem-se a organização inicial da implementação do modelo comportamental representado pelos diagramas de efeitos das funções adaptativas da Seção III. As próximas subseções discutem os refinamentos a partir deste ponto.

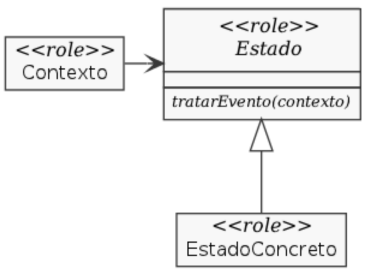

Figura 9. Estrutura do padrão de projeto *State* (DCL)

A. Estratégia inicial para o projeto de classes

O foco primário do paradigma de objetos é na organização das instruções do código-fonte [11], [24]. Seguindo a proposta de Gamma et al., pode-se modelar a topologia adaptativa do autômato por meio da classe TopAdapt. Em relação ao padrão *State*, esta classe corresponde ao papel Contexto. As diferentes configurações desta topologia serão modeladas como elementos do tipo *Config*, conforme ilustrado na Fig. 10. O modelo de implementação será derivado a partir de cada uma das configurações adaptativas especificadas na Tab. VI. Cada estado da especificação de requisitos comportamentais originará uma classe de implementação. Assim, para o modelo representado na Fig. 4 ter-se-á uma classe do tipo *Config* no modelo de objetos, oriunda dos estados de chegada das transições do estado **NãoAvaliado**.

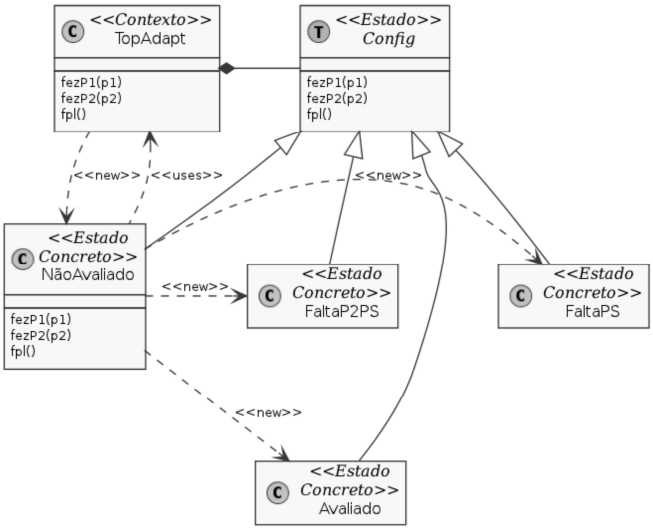

Figura 10. Organização inicial das classes do estudo de caso (DCL)

O tratamento da ocorrência de um evento do tipo fezP1 se inicia em um objeto da classe TopAdapt. A reação deste objeto é tal que ele encaminha uma mensagem envolvendo a operação fezP1(*P1*) para um objeto do tipo *Config*, mas da classe NãoAvaliado. Esta parte do modelo de objetos implementa a transição do estado **NãoAvaliado** para o estado **FaltaP2PS** da especificação adaptativa.

A execução do método adaptativo ma1P1 envolve uma transformação na estrutura do modelo de objetos: torna-se necessária a presença de uma variável para armazenar o valor da nota *P1* informada na ocorrência do evento fezP1. O objeto da classe NãoAvaliado instancia um FaltaP2PS e, em seguida, passa-lhe o valor de *P1* para armazenagem. Isso introduz uma dependência estática (de compilação de código-fonte) entre estas duas classes. Da mesma forma refina-se o modelo na ocorrência do evento fezP2.

A transição adaptativa fpl da configuração inicial apenas desencadeia a instanciação de um objeto da classe Avaliado , substituindo o objeto NãoAvaliado na ligação com o TopAdapt. Ou seja, realizada esta transição, apenas dois objetos passarão a existir na computação em andamento: um da classe TopAdapt e outro da classe Avaliado.

Por conseguinte, o diagrama de estados adaptativo da Fig. 5 é implementado por um objeto da classe TopAdapt, inicialmente conectado a um objeto da classe NãoAvaliado, mas do tipo *Config* . Na ocorrência de uma das transições fezP1 , fezP2 ou fpl, este objeto instancia um outro da classe FaltaP2PS, FaltaPS ou Avaliado, respectivamente, comandando a sua substituição por aquele recém instanciado. A substituição será realizada pelo objeto da classe TopAdapt. Como resultado, este objeto ficará conectado ao objeto que representa a nova configuração topológica.

Uma importante propriedade deste modelo se refere à encapsulação do efeito da transição e da adaptação nas classes descendentes. Por outro lado, as setas (de dependências de uso e de herança) revelam preocupações de acoplamento de instruções durante o refinamento do modelo. Em particular, a dependência cíclica entre TopAdapt e NãoAvaliado significa que o entendimento de uma classe está vinculado ao entendimento da outra (e vice-versa). O refinamento do modelo de implementação referente às classes FaltaP2PS e FaltaPS introduzirá novas dependências na direção de TopAdapt.

B. Arquitetura inicial que satisfaz à especificação adaptativa

O diagrama de classes UML apresentado na Fig. 11 foi produzido com a ajuda da ferramenta BlueJ em um projeto liderado por Kölling [25]. Há uma ênfase apenas nos relacionamentos de herança e de dependências entre as classes de implementação. Isso é suficiente para que sejam investigados os efeitos de mudança de instruções no código-fonte. A direção das setas indica a direção de dependência; a seta aponta para a classe que, se alterada, afetará a classe que se encontra na base da seta.

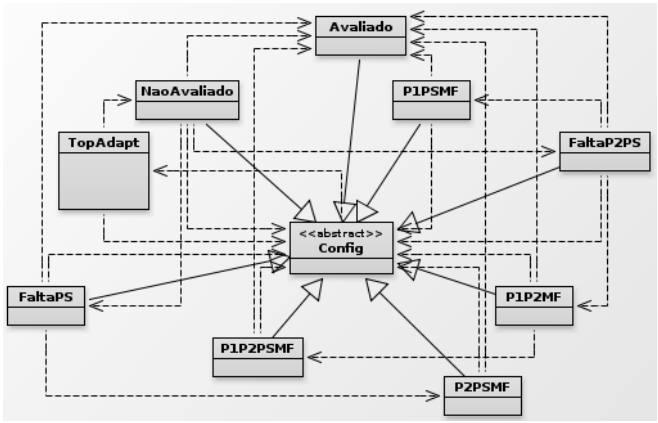

Figura 11. Arquitetura do modelo de implementação com *State* (BlueJ)

Observa-se que todas as classes descendentes de *Config* dependem de TopAdapt. A origem de tais dependências é que a configuração corrente deverá instanciar a configuração seguinte e passá-la para a topologia adaptativa (como um

objeto do tipo *Config*) de modo que o modelo de implementação reflita a topologia originalmente especificada.

Por outro lado, a dependência da classe TopAdapt para a NãoAvaliado se refere à instanciação do objeto que implementa a configuração inicial da topologia. TopAdapt instancia o objeto da classe NãoAvaliado, conectando-se a ele. Tal estrutura de objetos implementa uma configuração adaptativa inicial, conforme especificado no diagrama da Fig. 4.

Uma eventual edição de instruções da classe TopAdapt afetaria quais outras classes desta arquitetura? A ferramenta BlueJ usa um padrão de hachuramento para ajudar na visualização do alcance das alterações. Alterando-se a classe TopAdapt ela fica hachurada. Mas também ficarão hachuradas todas as demais classes que dela dependem (seja por uso, seja por herança). Na arquitetura até o momento elaborada, as notícias não são boas: todas as demais classes serão também hachuradas. A imagem da Fig. 12 mostra o padrão de hachuramento quando se altera alguma instrução da classe TopAdapt.

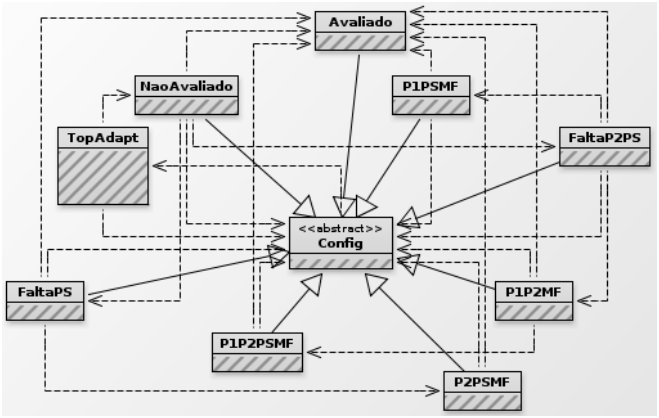

Figura 12. Classes afetadas por alterações em TopAdapt (BlueJ)

Devido a esta arquitetura de dependências, a edição de qualquer outra classe também provocará hachuramento generalizado. Tal propriedade resulta da dependência cíclica entre as classes de instruções deste modelo de implementação.

Sob o ponto de vista de controle de edições, tal arquitetura não é recomendável. Martin [10] propõe uma série de métricas para quantificação e crítica das propriedades de acoplamento e coesão. Neste trabalho, utilizar-se-á uma avaliação mais simples, baseada no padrão de hachuramento de classes do BlueJ: quanto maior a quantidade de classes hachuradas, pior é arquitetura de classes de implementação.

Como reorganizar as instruções desta implementação, ainda que satisfazendo à especificação adaptativa discutida na Seção III? O Princípio da Inversão de Dependências (DIP[3]) pode ajudar neste ponto.

[3] Dependency Inversion Principle

V. DESACOPLAMENTO POR DIP

Martin afirma que os ciclos de dependências podem ser interrompidos pela introdução de abstrações de desacoplamento [24]. O princípio DIP tem o seguinte enunciado:

> "(A) High level modules should not depend upon low level modules. Both should depend upon abstractions.
>
> (B) Abstractions should not depend upon details. Details should depend upon abstractions."[4]

A abstração *Topologia* resulta do emprego do DIP no modelo de classes até agora elaborado. Para ela são redirecionadas as dependências das classes que descendem de *Config*.

A. Refatoração

Nesta refatoração (no sentido de Fowler [26]), preserva-se a equivalência comportamental, embora a organização das instruções que compõem o programa tenha sido alterada. Em termos de manutenção de código, agora, alterações em TopAdapt não se propagam para as classes descendentes de *Config*. A abstração *Topologia* atua como uma barreira de alterações, impedindo que elas alcancem a classe NãoAvaliado. Por outro lado, alterações em NãoAvaliado propagam-se para TopAdapt. Tal dependência resulta da necessidade de instanciação do objeto que denota o estado inicial da configuração representado pelo diagrama da Fig. 13.

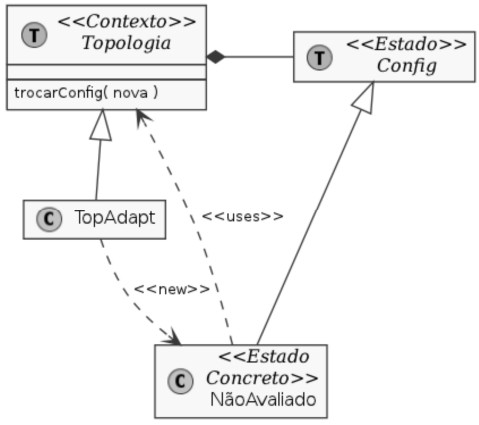

Figura 13. Desacoplamento de TopAdapt (DCL)

Com esta alteração arquitetural (introdução do tipo *Topologia*), o modelo de implementação assume a forma mostrada no diagrama da Fig. 14.

[4] Tradução livre: "(A) Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. (B) Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações."

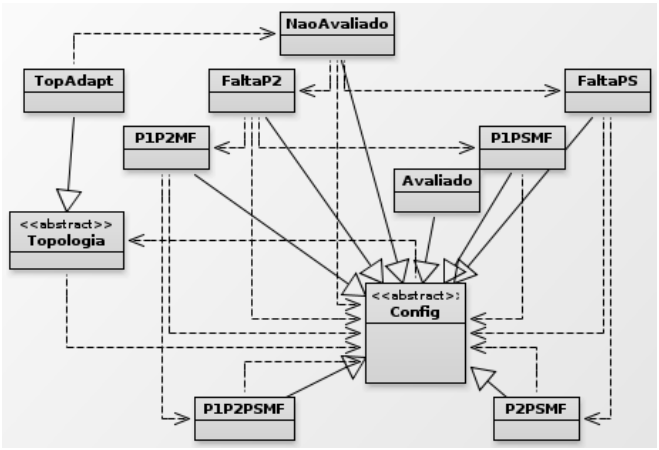

Figura 14. Revisão da arquitetura de classes com DIP (BlueJ)

B. Análise de alterações

O que acontece quando se altera alguma instrução da classe TopAdapt? Como nenhuma outra classe desta arquitetura depende de TopAdapt, apenas ela ficará hachurada. A Fig. 15 ilustra esta afirmação. Por outro lado, dependendo da classe descendente de *Config* que se altere, diversas outras ficarão hachuradas. Alguma mudança em instruções da classe NãoAvaliado, desencadearão alterações em TopAdapt: serão hachuradas. Entretanto, alterações na classe Avaliado, não provocarão hachuramento apenas nas classes *Topologia*, *Config* e Aprovado. Todas as demais dependem, direta ou indiretamente, da classe Avaliado.

Uma técnica complementar de desacoplamento pode ser empregada no sentido de manter constante a família de classes hachuradas por decorrência de edições: desacoplamento por *Factory Method*.

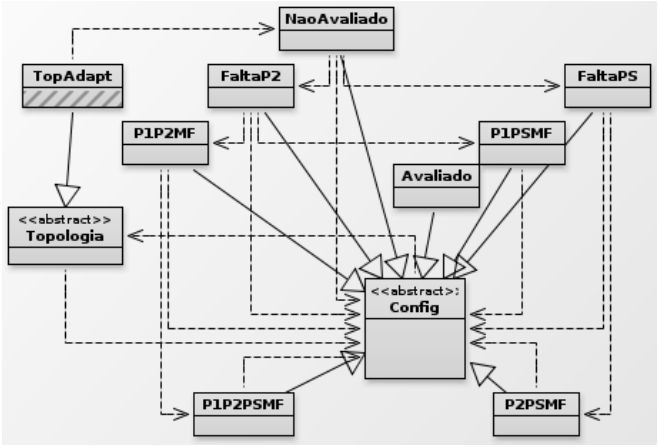

Figura 15. Alteração em TopAdapt (BlueJ)

VI. DESACOPLAMENTO POR FACTORY METHOD

Gamma et al. [23] propuseram três grupos de padrões de projeto: criação, estrutura e comportamento. Aqueles de criação procuram encapsular a lógica de instanciação de objetos em tempo de execução de modo que haja um isolamento entre a natureza e o uso objeto. Ou seja, a interação entre dois objetos exige que eles conheçam as suas naturezas (classes) ou é suficiente o conhecimento dos seus usos (tipos) para a realização de um passo computacional?

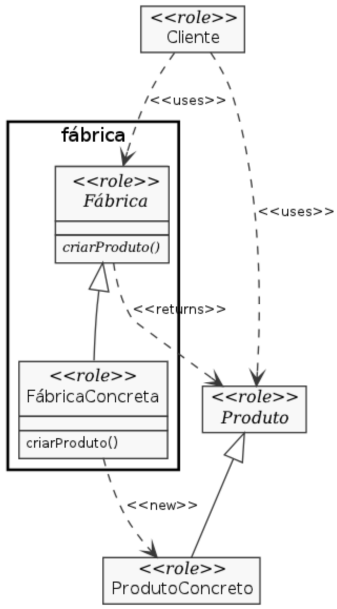

Figura 16. Estrutura de classes do Padrão *Factory Method* (DCL)

No problema em estudo, as classes que representam configurações de estados da topologia são acopladas em decorrência de precisarem instanciar as suas configurações sucessoras. Consequentemente, refinamentos em uma classe que se propõe a implementar uma particular configuração desencadeia revisões em classes que representam configurações anteriores. O padrão de projeto *Factory Method* procura resolver este problema por meio de uma interface de instanciação que redireciona a dependência para uma interface de uso, ao invés de uma interface de natureza (Fig. 16). Objetos da classe Cliente solicitam à *Fábrica* objetos que serão utilizados como *Produto* , não importando se eles terão a natureza de um ProdutoA ou ProdutoB .

A. Desacoplamento do espaço de configurações

O padrão de projeto *Factory Method*, aplicado ao problema tratado neste trabalho, resulta no modelo de classes destacadas no diagrama da Fig. 17.

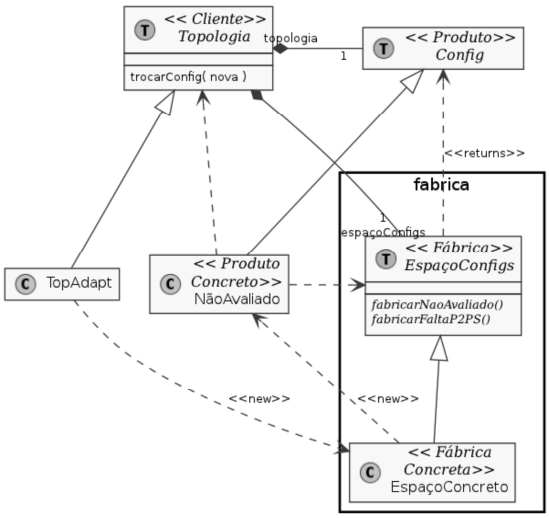

Figura 17. EspaçoConfigs como método de fabricação (DCL)

Em primeiro lugar, TopAdapt instancia um objeto de natureza EspaçoConcreto de configurações do tipo *EspaçoConfigs*. Deste modo, descendentes de *Config* (como é o caso de NãoAvaliado, por exemplo) poderão solicitar ao objeto do tipo *EspaçoConfigs* que instancie a próxima configuração da *Topologia*, sem ter uma dependência estática para isso.

Uma vez instanciado um novo objeto do tipo *Config* (por outro do mesmo tipo), ele é enviado para que a *Topologia* faça a troca da sua configuração. Por exemplo, considerando-se a configuração inicial do autômato, na ocorrência do evento fezP1, um objeto TopAdapt encaminha uma mensagem solicitando a realização da operação fezP1() para um objeto do tipo *Config*. Neste momento, o objeto que irá atender à mensagem é da classe NãoAvaliado. Ele solicita a um objeto do tipo *EspaçoConfigs* que instancie um objeto FaltaP2PS do tipo *Config*. Ele passa o valor da nota *P1* para este objeto e o encaminha para que a *Topologia* conduza a troca de estado da configuração.

O comportamento deste modelo de objetos é similar para o caso dos dois outros tipos de eventos detectáveis na configuração implementada pelos objetos da classe NãoAvaliado: fezP2 e fpl. Mas, supondo que a configuração implementada pela classe FaltaP2PS tenha sido atingida, como se refina este modelo de objetos? Quais transformações sofrerá a máquina de estados no caso das transições adaptativas? Como desenhar o comportamento adaptativo?

B. Evolução da arquitetura

No modelo de classes, a arquitetura até agora proposta indica como deverá ser feito o refinamento. Para detalhar o modelo de implementação da configuração referente ao estado **FaltaP2PS**, por exemplo, deve-se (Fig. 18):

1. adicionar as classes de implementação P1P2MF e P1PSMF;
2. indicá-las como sendo do tipo *Config*;
3. adicionar os métodos fabricarP1P2MF e fabricarP1PSMF na fábrica EspaçoConfigs.
4. implementar os métodos adaptativos ma2P2, ma2PS e ma2fpl na classe FaltaP2PS.

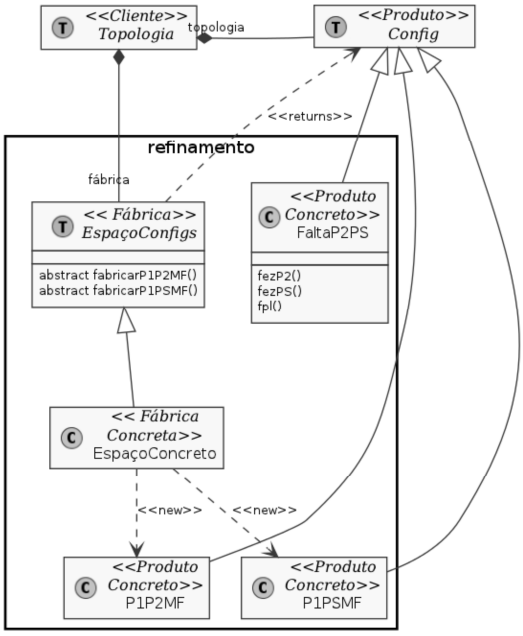

Figura 18. Refinamento do estado **FaltaP2PS** (DCL)

Um processo similar de refinamento ocorrerá para as demais classes que representam os estados da configuração adaptativa.

C. Modelo de Implementação com Factory Method

A versão da arquitetura contemplando o isolamento do mecanismo de instanciação é ilustrada no diagrama da Fig. 19. Observa-se que a arquitetura está se tornando gradualmente mais complexa no sentido de envolver cada vez mais elementos de implementação. Entretanto, o propósito fundamental das decisões de projeto é gerenciar a propagação das ondas de edição das instruções pelas classes de implementação.

A arquitetura ilustrada na Fig. 19 foi concebida de modo a organizar dois grupos de alterações: aquelas envolvidas nas instruções da classe TopAdapt e aquelas que se referem às descendentes de *Config*. As abstrações *Topologia* e *EspaçoConfigs* foram propostas exatamente para obter-se o seguinte efeito: (i) edições em um descendente de *Config* desencadeiam recompilações de TopAdapt e EspaçoConcreto; (ii) edições na classe TopAdapt não desencadeiam edições complementares de classes.

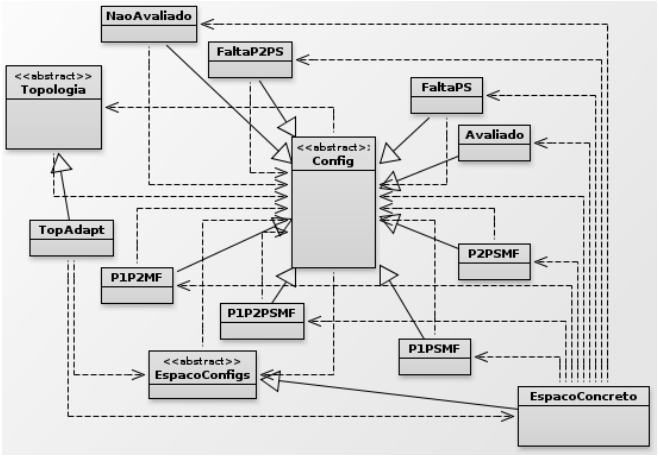

Figura 19. Isolamento de instanciação e desacoplamento (BlueJ)

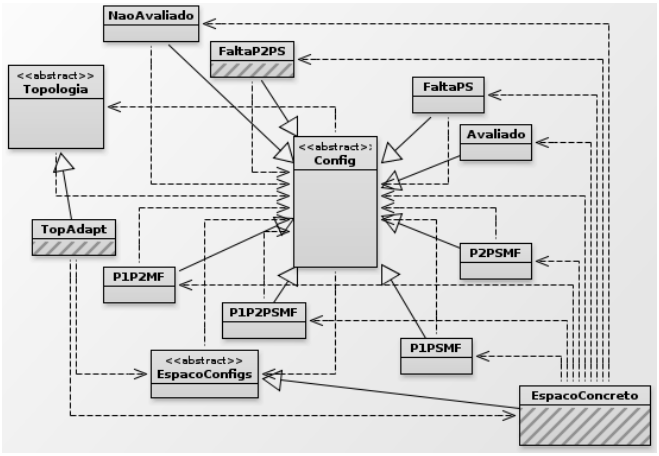

Figura 20. Alteração em FaltaP2PS (BlueJ)

O diagrama da Fig. 20 ilustra o padrão de hachuramento quando se edita a classe FaltaP2PS. Há revisão (ao menos de recompilação) apenas em outras duas classes arquiteturais.

O que acontece quando ocorre uma edição nas instruções da classe TopAdapt? Como as refatorações não afetaram as dependências em direção a esta classe, apenas ela será hachurada.

VII. CONCLUSÃO

Este trabalho de pesquisa teve foco no projeto de uma aplicação orientada a objetos, partindo de uma especificação adaptativa. A preocupação central lidou com a afirmação de Parnas a respeito da modularização e a sua relação com modelos de tempos de execução (modelos de computação). A questão básica da modularização considera fundamental o efeito dos critérios adotados para a decomposição em módulos.

Uma especificação adaptativa, proposta como resultado deste trabalho, suporta a organização do modelo comportamental de modo a indicar uma estratégia de modularização que enfatize as consequências de edição de código ao longo de uma arquitetura de software. Ou seja, classes de implementação são utilizadas para encapsular abstrações de funções adaptativas, originando métodos adaptativos. Desta forma, encapsulam-se as instruções que desencadeiam reconfigurações de topologias em classes de objetos, facilitando o entendimento do efeito da aplicação das funções adaptativas.

Especificações adaptativas, como introduzidas neste trabalho, podem ser sistematicamente produzidas a partir de tabelsaverdade, tabelas de decisão e tabelas de transição de estados. Tais técnicas consolidadas de programação, quando complementadas com métodos adaptativos (no sentido de representarem o efeito da aplicação de funções adaptativas), levam à elaboração de uma arquitetura de software que facilita a gerência das dependências estáticas.

O emprego de princípios de modelagem e de padrões de projeto também resultou em uma constatação: auxiliam na elaboração de uma arquitetura com dependências estáticas gerenciáveis. Ou seja, os princípios e padrões indicam pontos do modelo que são problemáticos quando se consideram as dependências estáticas.

O prosseguimento deste trabalho pode considerar as consequências de se desacoplar (no modelo de classses) a fábrica de métodos do espaço de configurações das edições da topologia de um autômato. Também pode-se investigar a geração automática da arquitetura de implementação referente aos métodos adaptativos. Isso automatizaria a geração de uma parte do código oriunda diretamentamente das tabelas de transição estendidas com abstrações de funções adaptativas.

APÊNDICE

Algumas das instruções do modelo de implementação do ensaio estão aqui apresentados. Ênfase foi dada para os serviços de instanciação dos objetos que representam as configurações do autômato e um detalhe de codificação do método adaptativo ma1P2.

EspacoConfigs — serviços de instanciação oferecidos pelos métodos de fabricação:

```
public abstract class EspacoConfigs {
  abstract Config fabricarNaoAvaliado();
  abstract Config fabricarFaltaP2PS();
  abstract Config fabricarFaltaPS();
  abstract Config fabricarAvaliado();
  abstract Config fabricarP1P2MF();
  abstract Config fabricarP1PSMF();
  abstract Config fabricarP2PSMF();
  abstract Config fabricarP1P2PSMF();
}
```

FaltaP2PS — implementação do método adaptativo ma1P2 e da ação semântica associados à transição de saída fezP2 do estado **FaltaP2PS**:

```
public class FaltaP2PS extends Config {
```

```
@Override
public void fezP2(Double p2) {
  // ma1P2
  Config proxima = fabrica.fabricarP1P2MF();
  configAdapt.trocarConfig(proxima);
  // semantic action
  proxima.definirP1(p1);
  proxima.definirP2(p2);
}
…
```


AGRADECIMENTOS

O autor agradece ao amigo e colega João J. Neto pelo apoio e incentivo para a realização deste trabalho. Suas ideias fundamentaram a proposta dos métodos adaptativos como uma técnica para refinar o módulo de controle de uma arquitetura de software. O autor também agradece aos dedicados membros do grupo de pesquisa GEMS-TIDD-PUCSP pelas valiosas discussões a respeito de técnicas e princípios para a modelagem com objetos. Finalmente, pelos importantes comentários e correções, o autor agradece aos anônimos revisores.

REFERÊNCIAS

[1] R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 7 ed., 2009. ISBN-13: 978-0073375977.

[2] P. Kruchten, "Architectural blueprints — the "4+1" view model of software architecture," *IEEE Software*, vol. 12, no. 6, pp. 42 – 50, 1995.

[3] "IEEE Standard glossary of software engineering termnology, std 610.12," 1990.

[4] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[5] B. P. Douglass, *Real-Time Design Patterns*. Object Technology Series, Reading, Massachusetts: Addison-Wesley, 2003.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," Sci. Comput. Programming, pp. 231–274, 1987.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[8] V. Setzer, *A construção de um compilador*. Campus, 1983.

[9] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, pp. 1–39, Publishing Company, 1993.

[10] R. C. Martin, *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice Hall, 1995. ISBN 0-13-203837-4.

[11] J. Lakos, *Large-Scale C++ Software Design*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-63362-0.

[12] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ.: Prentice-Hall, Inc., 1991. ISBN-13 978-0136298410.

[13] M. Jackson, "Representing structure in a software system design," *The Journal of Design Studies*, special issue on Studying Professional Software Designers, vol. 31, November 2010.

[14] M. Jackson, "Aspects of system description," in *Programming Methodology* (A. McIver and C. Morgan, eds.), p. 137160, Springer Verlag, 2003.

[15] M. Jackson, "Where, exactly, is software development?," in *Formal Methods at the Crossroads: from Panacea to Foundational Support; 10th Anniversary Colloquium of UNU/IIST* (B. K. Aichernig and T. Maibaum, eds.), (Lisbon), International Institute for Software Technology of The United Nations University, Springer-Verlag, March 18-21, 2002 2003. LNCS 2757.

[16] T. Rowlett, *The object-oriented development process: developing and managing a robust process for object-oriented development*. Prentice Hall PTR, 2001.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA:

[18] Addison-Wesley Longman Publishing Co., Inc., 2006. [18] M. V. M. Ramos, J. J. Neto, and Í. S. Vega, *Linguagens Formais: Teoria, Modelagem e Implementação*. Porto Alegre: Bookman, 1 ed., 2009.

[19] M. Fowler and K. Scott, *UML Distilled*. Addison Wesley, 2 ed., 2000.

[20] J. J. Neto, "Adaptive rule-driven devices — general formulation and case study," in *International Conference on Implementation and Application of Automata* (Springer-Verlag, ed.), pp. 234–250, 2001.

[21] Í. S. Vega, "An adaptive automata operational semantic," *Latin America Transactions, IEEE*, vol. 6, no. 5, pp. 461 – 470, 2009. ISSN 1548-0992.

[22] J. J. Neto, J. R. de Almeida Jr., and J. M. N. dos Santos, "Synchronized statecharts for reactive systems." Technical Report, 1998.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0201633612.

[24] R. C. Martin, *UML for Java Programmers*. Prentice Hall, 2003. ISBN 978-0131428485.

[25] D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall / Pearson Education, 4th ed., 2008. ISBN 0-13-606086-2.

[26] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN 0-201-48567-2.

Ítalo S. Vega recebeu os títulos de Mestre e Doutor em Engenharia de Software na Escola Politécnica da Universidade de São Paulo, Brasil, em 1993 e 1998, respectivamente. Professor Associado do Departamento de Computação da Pontifícia Universidade Católica de São Paulo e líder do Grupo de Estudos em Modelagem de Software (GEMS) do programa de pós-graduação em Tecnologia da Inteligência e Design Digital (TIDD).