Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Eduardo Ribeiro de Castro

Personalização de Produtos de Software baseada em Interfaces Orientadas a Aspectos

São Paulo
2012

Eduardo Ribeiro de Castro

Personalização de Produtos de Software baseada em Interfaces Orientadas a Aspectos

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para obtenção do título de Mestre em Engenharia de Computação.

Data da aprovação: ___/___/___

_____

Prof. Dr. Fernando Antônio de Castro Giorno (Orientador)

IPT - Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Membros da banca examinadora:

Prof. Dr. Fernando Antônio de Castro Giorno (Orientador)
IPT - Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Prof. Dr. Paulo Sergio Muniz Silva (Membro)
IPT - Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Prof. Dr. Marcos Ribeiro Pereira Barretto (Membro)
Escola Politécnica de Universidade de São Paulo

Eduardo Ribeiro de Castro

Personalização de Produtos de Software baseada em Interfaces Orientadas a Aspectos

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para obtenção do título de Mestre em Engenharia de Computação.

Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Fernando Antônio de Castro Giorno

São Paulo
Maio/2012

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT

C355p **Castro, Eduardo Ribeiro de**
Personalização de produtos de software baseada em interfaces orientadas a aspectos. / Eduardo Ribeiro de Castro. São Paulo, 2012.
100p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Fernando Antônio de Castro Giorno

1. Framework de customização 2. Customização de software 3. Programação orientada a aspectos 4. Interface de customização 5. Produto de software 6. Tese I. Giorno, Fernando Antônio de, orient. II. IPT. Coordenadoria de Ensino Tecnológico III. Título

12-75 CDU 004.416.3(043)

RESUMO

Produtos de *software* personalizáveis ou customizáveis oferecem a terceiros a possibilidade de alterar a seu funcionamento por meio de programação. A personalização pode ter como objetivo adicionar novas funcionalidades ou alterar as regras de negócio já presentes no produto. Para permitir tais alterações, o fabricante do software define *hot-spots* ou pontos de extensão, que são pontos onde o software faz chamadas a código escrito por terceiros. O fabricante deve prever quais funcionalidades precisarão ser personalizadas por algum cliente e deve criar os pontos de extensão nos locais adequados. Funcionalidades sem pontos de extensão expostos não podem ser personalizadas. Neste trabalho é proposta um *framework* de personalização baseado em programação orientada a aspectos. Os ajustes de personalização são codificados como aspectos e aplicados em pontos de extensão definidos pelo fabricante. O *framework* aceita a criação de pontos de extensão provisórios, permitindo que ajustes no produto seja aplicados mesmo em locais que não foram previstos pelo fabricante. Os pontos de extensão são disponibilizados na forma de interfaces orientadas a aspectos, evitando o problema da fragilidade dos pontos de corte. O *framework* é testado em aplicações criadas a partir de casos de personalização encontrados em produtos de uso corporativo.

Palavras-chave: Personalização; Customização; Programação Orientada a Aspectos; Interfaces Orientadas a Aspectos.

ABSTRACT

Software Product Customization based on Aspect-Oriented Interfaces

Customizable software are products that can be changed programmatically by its users or by a third party. Usually software customization involves adding new features or changing features that already exists in the product but don't fulfill completely the user needs. To make such changes possible, the manufacturer creates hotspots or extension points. The manufacturer must anticipate which features which need to be customized and create extension points in the appropriate places. Features without extension points cannot be customized. This paper proposes a customization framework based on aspect-oriented programming. Customized code is written as aspects, which are applied to extension points previously defined by the manufacturer. The framework supports the creation of provisional extension points, allowing that adjustments in the product are applied even in places that were not anticipated by the manufacturer. The extension points are provided in the form of aspect-oriented interfaces, avoiding the problem of fragile point cuts. Test cases for the framework reproduces customization situations found in some ERPs.

Keywords: Personalization, Customization, Aspect-Oriented Programming, Aspect-Oriented Interfaces.

Lista de Ilustrações

Figura 1 - Opções de modificação de Software (ROTHENBERGER, 2009). 7
Figura 2 - Efeito no esforço de manutenção (LIGHT, 2001) 12
Figura 3 - Classes do Editor Gráfico (KICZALES,2001) 18
Figura 4 - Comparação entre aspectos com e sem XPI 27
Figura 5 - EJPs aplicados a um *framework* (KULESZA, 2006) 29
Figura 6 - Comparação entre JPI e XPI. 33
Figura 7 - Elementos do *Framework* de Customização 45
Figura 8 - Uso de JPI no ACF 46
Figura 9 - Criação de ponto de extensão simples 48
Figura 10 - Criação de ponto de extensão complexo 48
Figura 11 - Equipes responsáveis por cada camada 51
Figura 12 - Aplicação de Teste 69

Lista de Tabelas

Tabela 1 - Opções de modificação de Software (BREHM, 2001). 10

Tabela 2 - Técnicas de personalização por código (SESTOFT, 2008). 15

Tabela 3 - Padrões de projeto aplicados à personalização (MICHAUD, 2003) 17

Tabela 4 - Classificação Estrutural dos Pontos de Extensão 62

Tabela 5 - Classificação Funcional dos Pontos de Extensão 62

Tabela 6 - Relação entre tipos estruturais e funcionais 64

Tabela 7 - Total de ajustes por tipo e subtipo. 67

Tabela 8 - Quantidade de serviços por tipo de operação 69

Tabela 9 - Customizações aplicadas ao corpo de teste 70

Tabela 10 - Alterações por evolução de produto 71

Tabela 11 - Ajustes devidos à inclusão de novo parâmetro 75

Lista de Listagens

Listagem 1 - Classes reescritas para gerarem mensagens 19
Listagem 2 - Geração de mensagens usando aspectos 20
Listagem 3 - Exemplo de código AspectJ 22
Listagem 4 - Exemplo de código @AspectJ 23
Listagem 5 - Exemplo de implementação de XPI em AspectJ 26
Listagem 6 - Exemplo de contrato de XPI em AspectJ 26
Listagem 7 - Exemplo de aspecto usando uma XPI 27
Listagem 8 - Código AspectJ de um EJP feito para o framework JUnit 30
Listagem 9 - Exemplo de Contrato do EJP 31
Listagem 10 - Exemplo de JPI (Interface de Ponto de Junção) 32
Listagem 11 - Tipos de ponto de junção e pontos de corte polimórficos 35
Listagem 12 - Eventos Declarativos em EScala (GASIUNAS, 2010) 36
Listagem 13 - Exemplo de Closure Joinpoint (BODDEN, 2011) 38
Listagem 14 - Exemplo de JPI 43
Listagem 15 - Exemplo de @JPI 43
Listagem 16 - Exemplo de chamada a hook 60
Listagem 17 - Exemplo de interface template de hook 61
Listagem 18 - Exemplo de classe concreta de hook 61
Listagem 19 - Exemplo de Hook de Substituição 85
Listagem 20 - Exemplo de Hook de Alteração 86
Listagem 21 - Exemplo de Hook Interno 87
Listagem 22 - Exemplo de Função (sem hooks) 88
Listagem 23 - Exemplo de Consulta 89
Listagem 24 - Exemplo de Edição de Registro 90
Listagem 25 - Exemplo de Rotina Genérica 91

Lista de Abreviaturas e Siglas

DAO	*Data Access Object*
EJP	*Extension Join Point*
ERP	*Enterprise Resource Planning System*
JPI	*Join Point Interface*
OA	Orientado(a) a Aspectos
OO	Orientado(a) a Objetos
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
REST	*Representational State Transfer*
SQL	*Structured Query Language*
XPI	*Cross-cutting Programming Interface*

SUMÁRIO

1	**INTRODUÇÃO**	**1**
1.1.	Motivação	1
1.2.	Objetivo	2
1.3.	Contribuições	2
1.4.	Método de Trabalho	2
1.5.	Organização do Trabalho	3
2	**FUNDAMENTAÇÃO TEÓRICA E ESTADO DA ARTE**	**5**
2.1.	Customização de Software	5
2.2.	Customização de Software via código	13
2.3.	Programação Orientada a Aspectos	17
2.3.1.	Introdução	17
2.3.2.	AspectJ	21
2.4.	Personalização e POA	24
2.4.1.	*Crosscutting Programming Interfaces* (XPI)	25
2.4.2.	*Extension Join Points* (EJP)	28
2.4.3.	*Join Point Interfaces* (JPI)	31
2.4.4.	*Join Point Types* e *Polymorphic Pointcuts*	34
2.4.5.	Eventos Declarativos Orientados a Objetos	35
2.4.6.	*Closure Joinpoints*	37
2.4.7.	Outras soluções	38
2.5.	Resumo	39
3	**PROPOSTA: O *FRAMEWORK* ACF**	**41**
3.1.	@JPI – *Join Point Interfaces* através de anotações	41
3.2.	Requisitos para o *Framework*	44
3.3.	Arquitetura do ACF	45
3.4.	Gerenciamento da Interface de Customização	49
3.5.	Construção do Protótipo	52
3.5.1.	Biblioteca de Anotações @JPI	52
3.5.2.	Biblioteca do ACF	52

4 TESTE DO PROTÓTIPO 55
4.1. Processo de customização 55
4.2. Tipos de Pontos de Extensão 59
4.2.1. Forma de Implementação - *Hooks* 60
4.2.2. Classificação dos Pontos de Extensão 61
4.3. Levantamento quantitativo de customizações 65
4.4. Teste do Protótipo ACF 68
4.4.1. Criação da Aplicação de Teste 68
4.4.2. Inserção das customizações 70
4.4.3. Evolução do produto 71
4.4.4. Ajustes das customizações 71
5 RESULTADOS DOS TESTES 72
5.1. Execução dos Testes 72
5.1.1. Inserção das Customizações 72
5.1.2. Inserção de customização não prevista no produto 74
5.1.3. Evolução da aplicação 74
5.2. Comparação entre ACF e *hooks* 76
5.2.1. Cobertura de Pontos de Extensão não previstos 76
5.2.2. Simplicidade do código 77
5.2.3. Robustez em relação à evolução do produto 78
5.2.4. Outras considerações 78
6 CONCLUSÕES E TRABALHOS FUTUROS 80
6.1. Conclusões 80
6.2. Trabalhos Futuros 80
REFERÊNCIAS 82
APÊNDICE A: EXEMPLOS DE USO DE HOOKS 85
APÊNDICE B: EXEMPLOS DE OPERAÇÕES CUSTOMIZÁVEIS 88

1 INTRODUÇÃO

1.1. Motivação

Um produto de *software* personalizável (ou customizável) é aquele que pode ser alterado de forma a incorporar funcionalidades que atendam necessidades específicas de cada cliente. As formas mais comuns de adaptação são por configuração ou por programação.

Quando a personalização é feita por meio de programação, o fabricante do produto normalmente oferece uma linguagem de programação e um *framework* com os quais os clientes podem escrever código para alterar o comportamento do produto. O fabricante também define pontos de extensão, que são pontos previamente preparados para receber o código de terceiros que estende o produto ou altera suas características.

O projeto do *software* precisa prever quais são as funcionalidades os clientes poderão querer alterar, e definir pontos de extensão para cada uma delas. Por maior que seja a experiência da equipe de projeto, dificilmente ela conseguirá prever as necessidades de customização de todos os clientes, especialmente se o produto for muito complexo ou tiver uma base instalada muito grande.

Quando é necessário customizar uma funcionalidade para a qual não há ponto de extensão, pode-se optar por permitir que o cliente altere diretamente o código fonte do produto. Essa opção é arriscada, pois o código criado pode introduzir erros no programa, comprometendo a qualidade do produto. Além disso, o código personalizado seria perdido quando o produto recebesse atualizações de versão.

Uma segunda possibilidade seria solicitar ao fabricante que alterasse o *software* para incluir novos pontos de extensão, em uma nova versão do produto. Mas essas alterações podem não ser do interesse do fabricante nem dos outros clientes, ou o tempo para entregá-las pode ser longo demais.

1.2. Objetivo

O objetivo deste trabalho é propor um *framework* de personalização de *software* que permita que um produto seja customizado mesmo em funcionalidades para as quais não foram previstos pontos de extensão. O *framework* deve permitir a criação de novos pontos de extensão sob demanda dos clientes, através de programação orientada a aspectos. Ele também deve oferecer ao fabricante do *software* uma forma de gerenciar esses novos pontos de extensão de forma a mantê-los compatíveis com versões futuras do produto.

O *framework* proposto é voltado para aplicações em três camadas (banco de dados, servidor de aplicação, navegador web ou aplicativo-cliente), construídos sobre a plataforma Java e o *framework Spring*.

1.3. Contribuições

A contribuição deste trabalho é uma proposta de uma arquitetura para customização de *software* baseada em Interfaces Orientadas a Aspectos. Outra contribuição é a criação de uma versão de JPI (*Joint Point Interface*) baseada em anotações Java que seja utilizável para definir interfaces orientadas a aspectos. Essa interface será usada como base para a construção do *framework* de customização.

1.4. Método de Trabalho

O trabalho será executado em cinco etapas.

a) Pesquisa Bibliográfica

A primeira etapa é a pesquisa sobre técnicas de adaptação de *software* e de programação orientada a aspectos. Em especial, trabalhos relacionados a interfaces orientadas a aspectos.

b) Especificação do *Framework*

A segunda etapa será a definição dos requisitos que o *framework* deve atender. Em seguida, será feita a especificação da arquitetura, baseada em interfaces orientadas a aspectos.

c) Criação do Protótipo

A terceira etapa será a criação de um protótipo. Será criada uma extensão do *framework* Spring que dê suporte à definição de interfaces orientadas a aspectos.

d) Aplicação do *Framework*

A quarta etapa será a aplicação do framework para personalizar fragmentos de código extraídos de *softwares* corporativos. Serão escolhidos trechos de código e tipos de ajuste que reproduzam os casos mais comuns de personalização. Os ajustes serão aplicados de duas formas: com as técnicas tradicionais baseadas em *hooks* e com o uso do *framework* proposto. Após a aplicação das personalizações, o produto original sofrerá alterações em seu código. O código personalizado também será alterado, caso isso seja necessário para que o produto continue funcionando.

e) Análise dos Resultados

A última etapa é a análise dos resultados. O objetivo é comparar as personalizações criadas com a abordagem tradicional e com a técnica proposta. Pretende-se identificar para quais tipos de adaptação a técnica proposta é mais ou menos eficiente.

1.5. Organização do Trabalho

A seção 2, Fundamentação Teórica e Estado da Arte, apresenta trabalhos atuais sobre o problema de adaptação de software, e as técnicas de POA que podem ser aplicadas nesse tipo de problema. Em particular, são descritos alguns padrões OO de adaptação e extensão de software, e técnicas de POA baseadas em interfaces OA.

A seção 3, *Framework* ACF, descreve o *framework* de personalização a ser criado. São apresentados os requisitos que ele deve atender e a arquitetura do sistema. Ao final da seção, há uma descrição da implementação do *framework*.

A seção 4, Teste do Protótipo, apresenta um levantamento dos tipos mais comuns de customização pedidos por clientes, para três produtos vendidos por um fabricante de software corporativo. Também mostra o uso do *framework* para personalizar trechos de código que representam os casos mais típicos.

A seção 5, Resultados dos Testes, compara a técnica proposta com o método tradicional de adaptação via *hooks*.

A seção 6, Conclusões, apresenta um resumo dos resultados do trabalho, identificando as contribuições, dificuldades encontradas e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E ESTADO DA ARTE

Esta seção é dividida em quatro partes. O item 2.1 apresenta uma visão geral do problema da customização de *software*. O item 2.2 trata das técnicas comumente utilizadas para o ajuste do *software* via alteração de código. O item 2.3 apresenta os conceitos básicos de programação orientada a aspectos (POA). O item 2.4 trata dos problemas encontrados no uso de POA para personalização de software e apresenta algumas técnicas encontradas na literatura que podem ser aplicadas para resolver ou reduzir esses problemas.

2.1. Customização de Software

Customização de Software (*Software Customization*) ou Personalização de Software é a prática de alterar a estrutura ou as funcionalidades de um produto de software para atender melhor às necessidades específicas de um usuário ou cliente. Numa situação ideal, o usuário (empresa ou indivíduo) compraria um produto que atendesse a todas as suas necessidades, sem necessidade de qualquer ajuste. Na prática, é comum que o usuário não consiga encontrar no mercado nenhum produto que atenda completamente aos seus requisitos. Isto é particularmente comum no caso de *softwares* corporativos, como ERPs (*Enterprise Resource Planning Systems*). Quando isso ocorre, a empresa adota o produto cujas funcionalidades estejam mais próximas do desejado, e faz ajustes pontuais onde o *software* não atende aos requisitos.

Zach (2011) e Rothenberger (2009) fizeram levantamentos dos principais motivos que levam as empresas personalizarem *softwares* corporativos. Algumas das motivações são culturais ou políticas, tendo pouca relação com as funcionalidades do produto. Exemplos: resistência à mudança; falta de conhecimento sobre o *software* no início da sua implantação; cultura organizacional; nível de experiência da equipe de implantação; baixa aceitação do *software* entre os usuários; dependência em relação empresas de implantação que incentivam a criação de customizações para aumentar sua receita.

Outras motivações têm fundamento técnico ou estratégico, que estão relacionados mais diretamente às características do produto que se quer alterar.

- Motivações técnicas. O *software* é comprado para substituir sistemas legados que estão ultrapassados tecnicamente, mas que atendem aos requisitos dos usuários. Nesse caso, é comum que se peçam ajustes para que o produto novo funcione de forma semelhante ao antigo.
- Processos de negócio únicos. *Softwares* corporativos normalmente pressupõem processos de negócio padronizados, que são usados pela maioria das empresas do mesmo ramo. Quando uma empresa possui um processo de negócio próprio que lhe traz uma vantagem competitiva, ela preferirá que o *software* seja alterado para se adequar a esse processo ao invés de mudar o processo para se adaptar ao *software*.
- Pequenas diferenças de funcionalidade. Procedimentos ou cálculos efetuados pelo produto podem ser um pouco diferentes daqueles usados pela empresa, exigindo alterações pontuais. Por exemplo, fórmulas para cálculo de valores em estoque, ou definição de políticas de preços.
- Maturidade da organização. Empresas em crescimento sofrem mudanças com frequência. À medida em que a empresa evolui, novos processos de negócio são criados, e o *software* deve ser adaptado a eles.
- Maturidade do *Software*. Se o produto for relativamente novo ele pode não conter todas as funcionalidades exigidas pelo cliente. Nesse caso, é possível que sejam feitas customizações, que serão substituídas por novos módulos do produto em versões futuras.
- Mudança nas expectativas dos usuários. Depois que os usuários começam a usar o sistema, eles passam a conhecer melhor a sua filosofia e suas funcionalidades. Com isso, passam a solicitar modificações para melhorar a forma de operação.

Quando o sistema não adere totalmente às necessidades do usuário, há algumas formas de adaptar o produto ou o processo para cobrir essa discrepância. A figura 1 mostra como Rothenberger sumarizou essas opções. Cores mais escuras

na figura indicam maior probabilidade de que as modificações devam ser refeitas depois de uma atualização de versão do produto.

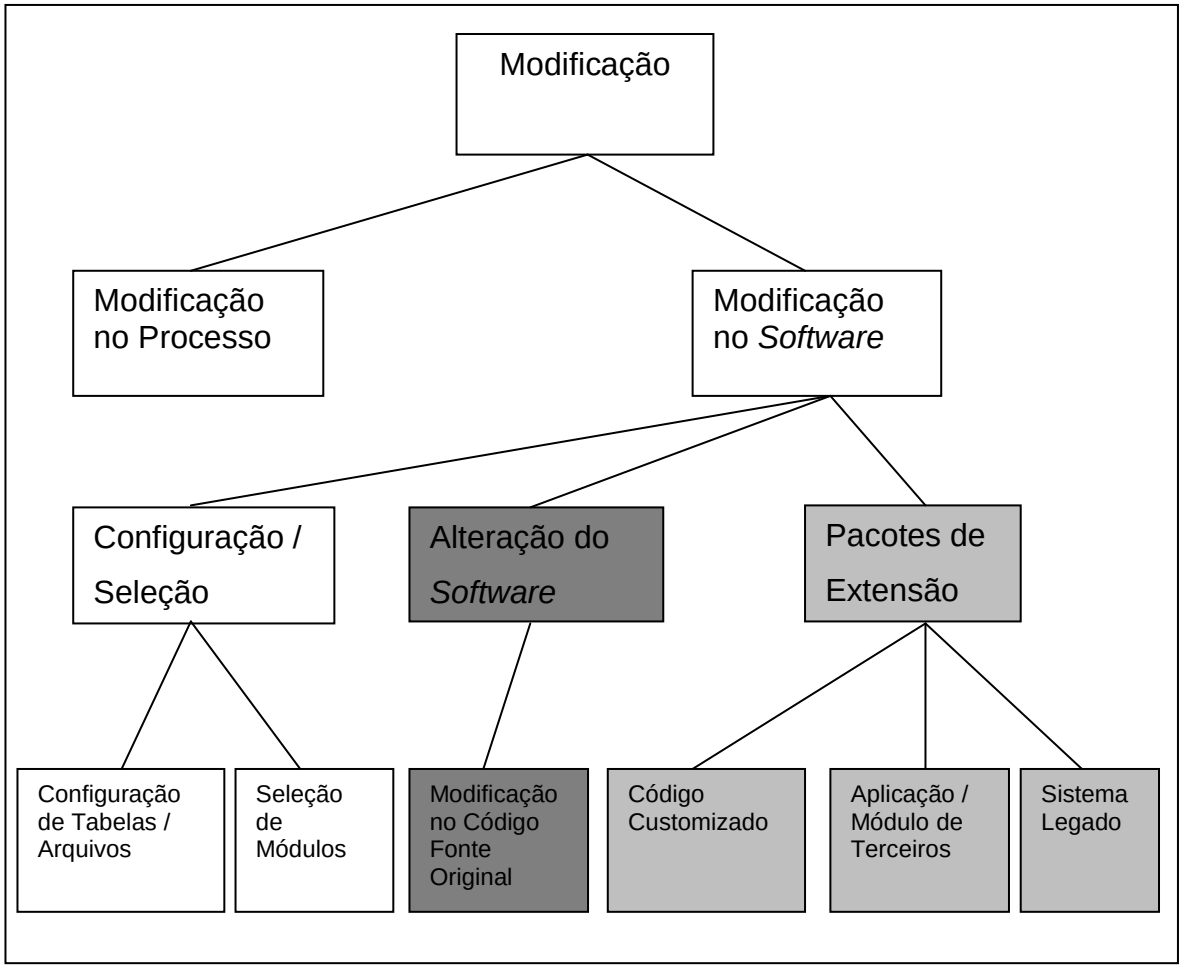


Figura 1 - Opções de modificação de Software (ROTHENBERGER, 2009).

A escolha principal é entre modificar o processo ou modificar o *software*. Modificação no processo é quando o usuário ou empresa muda sua forma de trabalhar para se adequar ao sistema. Esta é a opção recomendada por fabricantes de *ERPs*, por exigir menos esforço de adaptação do produto durante a implantação. A outra opção é por modificações no *software*. Se por alguma das razões citadas anteriormente o cliente decide adaptar o produto, há três opções possíveis: ajustes de configuração/seleção, pacotes de extensão, e alteração do *software*.

a) Configuração/Seleção. É a opção mais simples de modificação. Não exige criação ou alteração de código. Depende somente de funcionalidades já

existentes no produto, que são ajustadas para atender os requisitos do usuário.

- Configuração: é basicamente uma parametrização do sistema. Pode ser feita através de alterações de arquivos ou tabelas.
- Seleção de módulos: é a escolha de quais módulos do produto serão implantados. Escolhem-se os que sejam mais apropriados para atender aos requisitos.

b) Pacotes de Extensão *(Bolt-ons)*. *S*ão pacotes ou componentes projetados para complementar a funcionalidade do produto. São construídos para interagir com o produto através de interfaces de programação (APIs) criadas especificamente com esse propósito. Também podem interagir com o produto através do banco de dados ou de troca de arquivos. Não precisam ser necessariamente construídos na mesma plataforma que o software original, embora isto seja muitas vezes recomendável. Podemos classificar os pacotes de extensão em três categorias:

- Módulos de Terceiros. São produtos criados por outras empresas e comprados para complementar o *software* original. O objetivo pode ser adicionar novas funcionalidades ou substituir algum módulo do produto por outro mais adaptado à necessidade do cliente.
- Sistemas Legados. Aplicações pré-existentes que são adaptadas para trabalhar em conjunto com o *software* corporativo.
- Código customizado. É um pacote criado por implantadores ou usuários para resolver um problema específico de um cliente/usuário.

Como os pacotes de extensão dependem de interfaces oferecidas pelo *software* ou da estrutura do seu banco de dados, e como essas interfaces podem mudar com a evolução do produto, podem ser necessários ajustes nos códigos dos pacotes de extensão quando se faz uma atualização de versão do produto. Mas os fabricantes normalmente tentam manter as interfaces estáveis ao longo das versões, e mesmo quando as alteram podem fazê-lo de forma planejada, avisando sua base de usuários com bastante antecipação.

c) Modificação no código fonte. Nesse caso, o código do produto original é alterado *in loco* para atender às necessidades de um cliente específico. Esta é a alternativa mais complexa, pois exige um esforço muito maior de programação e um conhecimento mais detalhado da arquitetura do produto. Além disso, todas as alterações deverão ser revisadas, testadas, corrigidas e reaplicadas ao produto cada vez que uma nova versão for instalada.

Brehm et al. (2001), apresentam outra classificação das opções de modificação de software, mais voltada aos elementos que fazem parte de um ERP. Nesse caso, consideraram-se sistemas tradicionais em três camadas:

- Camada de comunicação: faz a comunicação com os usuários (através de interfaces gráficas) e com outros sistemas
- Camada de aplicação: contém as rotinas do sistema. Cálculos, *workflows*, etc.
- Camada de banco de dados: gerencia os dados do sistema. Tradicionalmente é um sistema de banco de dados relacional

A tabela 1 mostra a classificação de Brehm. A ordem dos itens indica o grau de impacto da modificação, isto é, o quanto a estrutura do ERP é afetada por ela. Os itens do final da tabela são os de impacto maior.

Os dois primeiros itens da lista, “Configuração” e “Pacotes de Extensão”, são os mesmos citados por Rothenberger. O último item, “Modificação de código fonte”, também é citado, sendo considerado o tipo de ajuste de maior impacto. Os outros itens possuem complexidade intermediária. Eles exigem certo esforço de programação, mas não afetam a estrutura do produto. Edição de telas, edição de relatórios, programação do ERP ou de workflows são feitas com linguagens de programação disponibilizadas pelo fabricante do produto, que não necessariamente são a mesma usada para construir o produto. *User exits* e interfaces externas exigem a criação de rotinas que interagem com o produto seguindo uma especificação de interface pré-estabelecida pelo fabricante do produto, cuja

evolução é feita de forma planejada. Podem ser escritas em linguagem nativa ou com a linguagem disponibilizada pelo fabricante.

Tabela 1 - Opções de modificação de Software (BREHM, 2001).

Modificação	**Descrição**	**Camadas afetadas**
Configuração	Definição de parâmetros do sistema	Todas
Pacotes de extensão (*Bolt-ons*)	Pacote de terceiros projetado para adicionar funcionalidades específicas ao ERP.	Todas
Máscaras de Telas	Editar as telas para entrada e saída de dados.	Comunicação
Relatórios	Criação de novos relatórios ou alteração dos existentes.	Aplicação, Banco de Dados
Programação de *Workflows*	Modificação de *workflows* padrão do sistema ou criação de novos.	Aplicação, Banco de Dados
User exit	Sub-rotina chamada durante a execução de evento pré-definido. O cliente pode usá-la para substituir uma sub-rotina do sistema por sua própria versão.	Aplicação, Banco de Dados
Programação do ERP	Programação de aplicações adicionais, sem mudar o código fonte do sistema (usando a linguagem de programação do fabricante).	Todas
Desenvolvimento de Interfaces Externas	Programar interfaces entre o ERP e sistemas legados ou produtos de terceiros.	Aplicação, Banco de Dados
Modificação do código fonte	Mudanças no código fonte do produto, desde pequenas mudanças pontuais até alterações em módulos inteiros.	Todas

A literatura sobre customização de produtos (VAUCOULEUR,2009), (BREHM, 2001), (ZACH,2011), (MICHAUD, 2003) normalmente identifica três agentes principais envolvidos no processo de adaptação do software: o fabricante, o implantador e o cliente ou usuário final.

- Fabricante: é a empresa que projeta e constrói o produto. No projeto, o fabricante deve prever quais serão os pontos onde provavelmente haverá

necessidade de customização e criar a infraestrutura que vai tornar essas alterações possíveis. Ele também deve decidir se deixará o código do produto aberto para que implantadores ou clientes possam alterá-lo. Em resumo, é o fabricante que decide como o produto poderá ser customizado.

- Implantadores: também chamados consultores, customizadores ou parceiros. É a equipe que implanta o produto, adaptando-o aos requisitos do cliente quando necessário. Normalmente, trata-se de uma empresa especializada nesse tipo de atividade. Mas em algumas situações, pode ser uma divisão da empresa fabricante ou uma equipe do próprio cliente. Os implantadores devem ter conhecimento detalhado das funcionalidades e da estrutura do produto e também das necessidades do cliente e de seus processos de trabalho.
- Cliente: é a empresa que compra e usa o produto. É dela que partem os requisitos de customização. O cliente trabalha em conjunto com os implantadores na definição dos requisitos e acompanha a implantação.

Há dois momentos distintos nos quais surge a necessidade de customização (ZACH,2011): durante a implantação ou durante a operação.

- Implantação: fase na qual o produto é avaliado, instalado e testado, e os usuários são treinados. Este é momento no qual os requisitos iniciais do cliente são levantados e é tomada a decisão sobre quais funcionalidades precisarão ser adaptadas. Nesta etapa, são feitos ajustes sem os quais o sistema não pode entrar em operação. Algumas das motivações para customização nessa etapa podem ser: (a) a empresa adota processos de negócio únicos, sem equivalente no produto e que lhe dão uma vantagem competitiva; (b) há a necessidade de integração com sistemas legados; (c) pequenos diferenças na forma como cálculos ou outras rotinas são feitas exigem ajustes pontuais.
- Operação: quando o produto já está em uso. Mesmo que não haja falhas no processo de implantação e todos os ajustes necessários tenham sido previstos, pode surgir a necessidade de novas customizações. Algumas

razões para isso: (a) evolução da empresa, que leva à alteração de seus processos de trabalho; (b) evolução do produto, que afeta as customizações já existentes ou passa a exigir novos ajustes; (c) evolução dos usuários, que ao se familiarizarem com o produto podem surgir com novos requisitos.

Customizações exigem manutenção. Correções de *bugs*, documentação, adaptação a novas versões do produto, ou adaptação a novas demandas dos usuários são algumas formas de manutenção. Um dos principais motivos para uma empresa comprar um software corporativo ao invés de construí-lo é o fato de que com isso a manutenção e o desenvolvimento do sistema passam a ser de responsabilidade do fabricante, o que reduz custos. Mas ao optar por customizar o produto, o cliente volta a ser responsável pela manutenção de parte do sistema, aquela que foi ajustada. Mesmo que ela terceirize esse trabalho para os consultores ou implantadores, ela ainda terá um custo maior do que se não alterasse o sistema.

Light (2001) fez uma pesquisa em algumas empresas para descobrir quais as consequências que as customizações em ERPs trazem do ponto de vista de manutenção. A figura 2 relaciona os tipos de modificação e seu efeito no esforço de manutenção.

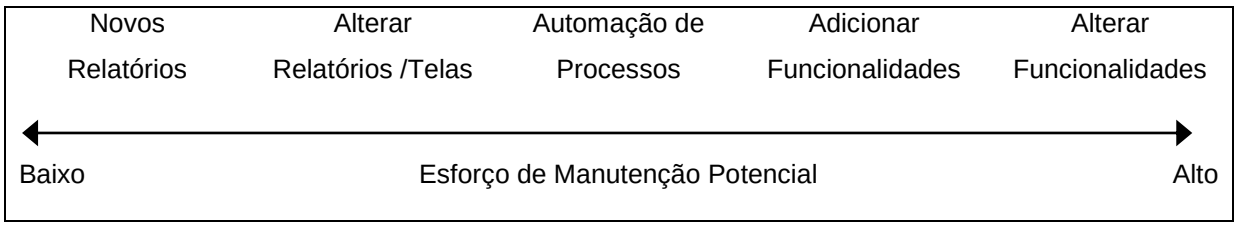


Figura 2 - Efeito no esforço de manutenção (LIGHT, 2001)

A alteração e o acréscimo de funcionalidades são ajustes que envolvem modificação no código fonte, e por isso são os que trazem maior impacto do ponto de vista do custo de manutenção.

2.2. Customização de Software via código

Sestoft (2008) apresentou um levantamento de técnicas e elementos de linguagem de programação usadas na personalização de software. O objetivo foi classificar as técnicas em função de sua maior estabilidade em relação à evolução do software. As técnicas avaliadas foram:

- Herança: uso de herança para estender ou alterar funcionalidades.
- Interfaces: técnica para esconder informação. Programação orientada a interfaces.
- Polimorfismo paramétrico: uso de tipos genéricos (*generics* em Java, *templates* em C++).
- Eventos Síncronos/*Callbacks*: eventos disparados em certos momentos do processamento da aplicação, que chamam funções customizadas (passando parâmetros).
- Métodos Parciais: característica da linguagem C#, que permite a adição de métodos em uma classe, sendo uma alternativa aos eventos síncronos.
- *Mixins*/*Traits*: estruturas permitidas por algumas linguagens, como Scala, que resolvem alguns problemas ligados à herança única.
- Aspectos: programação orientada a aspectos. Usada para injetar código adicional em um programa, ou substituir trechos de código em tempo de compilação ou execução.
- Linhas de produtos de software: construção personalizada de produtos a partir de componentes básicos pré-construídos.
- Camadas: técnica usada no sistema Microsoft AX, no qual as funcionalidades são distribuídas em oito camadas. A mais baixa é o núcleo do produto, e cada camada acima é um nível de personalização mais próximo do usuário final.

Quatro critérios foram usados para avaliar as técnicas acima.

a) Necessidade de antecipar onde e quais serão as alterações.

   Há três formas de antecipação dos ajustes:

   - **Pontos de extensão**: quando o arquiteto do produto define de antemão em que pontos do código fonte a personalização pode ser aplicada.
   - **Tipos de extensão**: quando se requer a antecipação do conteúdo das personalizações.

- **Nenhuma antecipação:** quando não é necessário prever onde os ajustes serão aplicados.

b) Controle sobre a personalização.

É necessário que o arquiteto do produto imponha algumas restrições à forma como os ajustes são criados e chamados pelo sistema. Assim, ele pode impedir que ajustes feitos por terceiros afetem a coerência e a correção da implementação padrão. Os tipos de controle são:

Em projeto: significa que as restrições são definidas durante a etapa de projeto do *kernel* do produto.

Em tempo de execução: indica que a técnica permite ativação, e controle dos ajustes enquanto o produto está sendo executado.

Nenhum controle: indica que a técnica não provê nenhuma forma de controle sobre o código personalizado.

c) Resiliência à evolução no produto.

Resiliência é uma indicação de como a evolução do produto pode afetar o código personalizado. Quanto maior a resiliência, maior a chance de os ajustes continuarem funcionando mesmo se o produto sofrer atualizações de versão. Há três classificações: **nenhuma**, **restrita** e **alguma**, em ordem crescente de resiliência.

d) Suporte a múltiplas personalizações no mesmo ponto.

Pode ser sequencial, paralelo ou sem suporte.

Paralelo: significa que múltiplos ajustes podem ser aplicados ao mesmo ponto, mesmo sendo feitos por desenvolvedores independentes.

Sequencial: quando há uma ordem na qual os ajustes são construídos, havendo uma dependência estrutural entre eles.

Sem suporte: só um ajuste de personalização é permitido no mesmo ponto.

A tabela 2 mostra as técnicas de adaptação classificadas segundo esses quatro critérios.

Tabela 2 - Técnicas de personalização por código (SESTOFT, 2008).

Técnica	**Necessidade de Antecipar as alterações**	**Controle sobre as alterações**	**Resiliência à evolução do produto**	**Suporte a múltiplas personalizações**
Herança	Pontos de extensão	-No projeto	restrita	Seqüencial
Interfaces	Pontos de extensão	-No projeto	restrita	Sem suporte
Parapolimorfismo	Pontos de extensão	-No projeto	restrita	Seqüencial
Eventos/ *Callbacks*	Pontos de extensão	-No projeto -Tempo de execução	restrita	Paralelo
Métodos Parciais	Pontos de extensão	-No projeto	restrita	Sem suporte
Mixins, *traits*	Pontos de extensão	-No projeto	restrita	Paralelo
Aspectos	Nenhuma	-Nenhum	nenhuma	Paralelo
LPS	Tipos de extensão	-No projeto	alguma	Sem suporte
Camadas	Nenhuma	-No projeto	nenhuma	Sequencial

Em comparação com as demais técnicas, o uso de POA apresenta vantagens e desvantagens. A maioria das técnicas exige que o arquiteto do produto defina de antemão os pontos de extensão do sistema. Alterações feitas usando aspectos são as que proporcionam maior liberdade ao customizador, ao dispensar os pontos de extensão. Também permitem múltiplas alterações independentes no mesmo ponto de código.

Por outro lado, o uso de POA é mais frágil do ponto de vista de robustez em relação a mudanças do produto, já que depende fortemente da assinatura dos métodos e das classes base. POA também oferece ao fabricante do produto poucos mecanismos para restringir as personalizações.

Seis das técnicas apresentadas acima são na verdade características de linguagens orientadas a objetos: Herança, Interfaces, Parapolimorfismo, Eventos/*Callbacks*, Métodos Parciais e *Mixins/Traits*. Todas elas se mostram mais vantajosas que POA no que se refere à resiliência (restrita) e ao controle sobre as alterações (Em projeto). Mas todas restringem a liberdade de personalizar o código ao exigirem a definição de pontos de extensão.

APIs de personalização baseadas em POO normalmente adotam alguns dos padrões de projeto apresentados por Gamma et al. (1995). Michaud (2003) realizou um levantamento dos padrões de projeto mais usados na construção dessas APIs. A tabela 3 mostra o contexto no qual eles são usados.

O padrão *Strategy*, por exemplo, pode ser usado para permitir ao aplicativo selecionar entre várias opções de implementação de uma funcionalidade. A adição de uma nova estratégia pode ser feita com o uso do padrão *Abstract Factory*. O padrão *Decorator* é usado tanto para estender como para restringir uma funcionalidade. Os padrões *Composite*, *Decorator* e *Command* são típicos de *toolkits* de interface gráfica.

Todos esses padrões, estruturalmente, acabam exigindo a definição de pontos de extensão. Por exemplo, o padrão *Strategy* (Gamma et al., 1995) usado na seleção de funcionalidades é composto de três classes: *Context*, *Strategy* e *ConcreteStrategy*. As classes *Context* e *Strategy* são parte do produto, e devem ser previstas no projeto, sendo o ponto de extensão. Só a classe *ConcreteStrategy* é código escrito pelo implantador. O conjunto de todos os pontos de extensão é o que forma a API de personalização. A API está acoplada ao *kernel* do produto. O *kernel* deve ser instrumentado para expor parte de sua funcionalidade para ajustes de personalização.

Com a evolução do produto, a estrutura interna do *kernel* pode mudar, afetando a API e forçando sua mudança também. Isto pode ser tratado de duas formas. Ou a API é alterada para expor essas mudanças para o implementador, quebrando a compatibilidade com o código já existente, ou são criados adaptadores no código interno do API que ajustem a estrutura nova do *kernel* à interface antiga

oferecida aos implementadores. Desta forma, fica a cargo do projetista do produto a decisão de quebrar ou não a compatibilidade do código de personalização.

Tabela 3 - Padrões de projeto aplicados à personalização (MICHAUD, 2003)

O que é personalizado	**Padrões de projeto**	**Propósito.**
Apresentação: Arquitetura de informação	*Bridge*	Desacopla Interface gráfica da implementação
Apresentação: Projeto Gráfico	*Composite, Decorator, Command*	*Toolkits* de componentes gráficos auxiliam a construção de telas
Dados: Formato	*Builder, Interpreter*	Componentes para adaptar e atualizar formatos de dados
Dados: Conteúdo	-	-
Controle: Seleção de Funcionalidades	*Strategy*	Composição de Componentes
Controle: Acréscimo de Funcionalidades	*Abstract Factory*	Composição de Componentes
Controle: Melhoria de Funcionalidades	*Strategy, Decorator*	Adaptação de Componentes
Controle: Restrição de Funcionalidades	*Decorator*	Adaptação de Componentes
Controle: Coordenação de Funcionalidades	*Mediator, Adapter*	Composição de Componentes
Controle: especificação de Funcionalidades	*Template Method, State*	Composição de Componentes

2.3. Programação Orientada a Aspectos

2.3.1. Introdução

A Programação Orientada a Aspectos (POA) foi proposta por Kiczales et al. (1997) como uma forma de melhorar a modularização de sistemas escritos em linguagens orientadas a objetos. O objetivo é reduzir ou eliminar a repetição de código de programação de responsabilidades transversais (*crosscutting concerns*).

Responsabilidades transversais são funcionalidades auxiliares tais como auditoria, controle de transações e segurança, cuja implementação normalmente é repetida e espalhada em várias classes. A proposta de POA é reunir o código associado a essas funcionalidades em métodos (funções) separados das classes. Na compilação ou durante a execução do software, o código desses métodos é injetado em todos os métodos das outras classes onde auditoria, controle de transações e outras responsabilidades transversais são exigidos.

O exemplo a seguir, extraído de (KICZALES, 2001), mostra o uso de AspectJ em um de editor gráfico 2D hipotético. A Figura 3 contém o diagrama de classes deste programa.

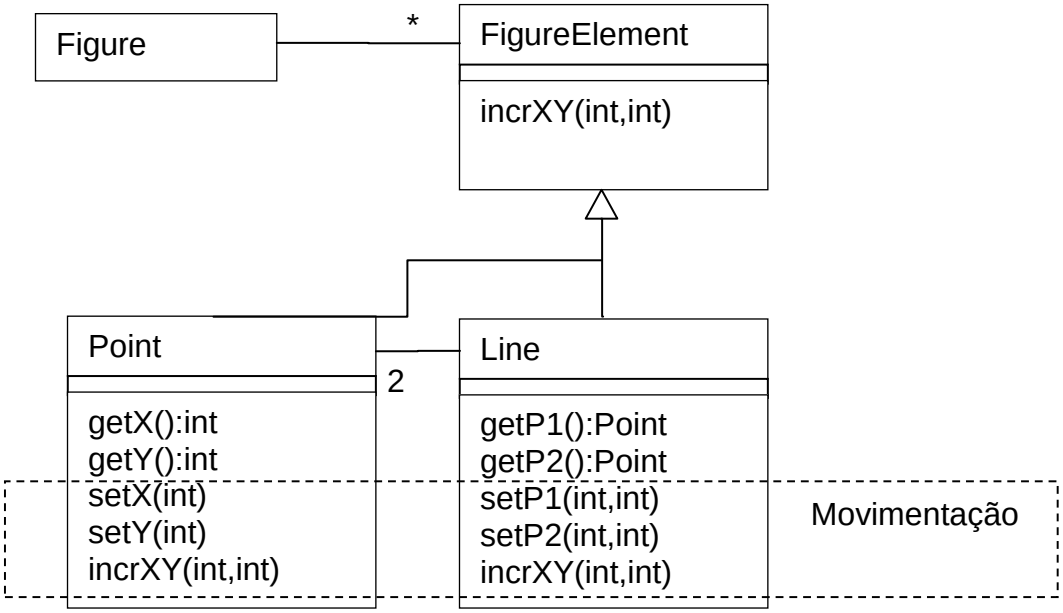


Figura 3 - Classes do Editor Gráfico (KICZALES,2001)

A classe `Figure` é uma coleção de elementos (classe `FigureElement`), os quais podem ser pontos ou linhas (classes `Point` e `Line`). O método `incrXY(int,int)` serve para mudar a posição dos elementos, adicionando valores às coordenadas x e y. Os métodos `setX(int)`, `setY(int)`,`setP1(int,int)` e `setP2(int,int)` servem para mudar a posição absoluta das coordenadas x e y.

Pode surgir a necessidade de adicionar uma nova funcionalidade ao editor, que seria gerar mensagens no console de erros cada vez que um elemento gráfico

fosse movido. Nesse caso seria preciso alterar todos os métodos que afetam o posicionamento das figuras incluindo o código de impressão de mensagens. A listagem 1 mostra como seriam essas alterações.

```
01 class Point {
02   ...
03   public void setX(int x) {
04     //Código adicionado para geração de mensagem
05     System.err.println("Elemento movido");
06     ...
07   }
08
09   public void setY(int Y) {
10     //Código adicionado para geração de mensagem
11     System.err.println("Elemento movido");
12     ...
13   }
14
15   public void incrXY(int x,int Y) {
16     //Código adicionado para geração de mensagem
17     System.err.println("Elemento movido");
18     ...
19   }
20 }
21
22
23 class Line {
24   ...
25   public void setP1(int x,int Y) {
26     //Código adicionado para geração de mensagem
27     System.err.println("Elemento movido");
28     ...
29   }
30
31   public void setP1(int x,int Y {
32     //Código adicionado para geração de mensagem
33     System.err.println("Elemento movido");
34     ...
35   }
36
37   public void incrXY(int x,int Y) {
38     //Código adicionado para geração de mensagem
39     System.err.println("Elemento movido");
40     ...
41}
42 }
```

Listagem 1 - Classes reescritas para gerarem mensagens

O mesmo código é repetido em vários métodos e em várias classes. É o que se chama de **espalhamento** (*scattering*). Se no futuro houver a necessidade de mudar a forma de geração de mensagens, todos esses pontos de código teriam que ser alterados.

Outro problema com este código é o **emaranhamento** (*tangling*). É quando um único método possui código para tratar de dois ou mais objetivos distintos, o que torna o objeto menos coeso. No exemplo, o código do método `incrXY(int,int)` seria responsável por mudar a posição do elemento, que é sua funcionalidade principal, e por gerar *logs*, uma função acessória. Quando o código é muito emaranhado, sua manutenção se torna mais difícil.

A programação orientada a aspectos propõe uma forma de isolar o código de responsabilidades transversais em um módulo separado. Este código é copiado, ou injetado, em todos os pontos necessários durante a compilação do *software*. O objetivo é reduzir o emaranhamento e o espalhamento. A listagem 2 mostra como seria a geração das mensagens através de aspectos.

```
01 aspect Logging
02 {
03
04     pointcut move():
05     call(void FigureElement.incrXY(int,int)) ||
06         call(void Point.setX(int))              ||
07         call(void Point.setY(int))              ||
08         call(void Line.setP1(Point))            ||
09         call(void Line.setP2(Point));
10
11
12     after(): move() {
13         System.err.println("Elemento movido");
14     }
15 }
```

Listagem 2 - Geração de mensagens usando aspectos

A linha 1 é declaração de um **aspecto**, de nome `Logging`. O aspecto (*aspect*) é a unidade básica de modularização em POA. As linhas 4 a 9 contêm a definição do **ponto de corte** `move`. Ponto de corte (*pointcut*) é a definição de onde será injetado o código dos aspectos. Este ponto de corte `move` consiste em cinco **pontos de junção**, que são as chamadas aos métodos `FigureElement.incrXY(int,int)`, `Point.setX(int)`, `Point.setY(int)`, `Line.setX(int)` e `Line.setY(int)`.

O ***advice*** é definido nas linhas 12 a 14. O *advice*é o código a ser injetado. A palavra chave `after` indica que ele deve ser executado imediatamente depois de qualquer chamada feita pelo programa a cada um dos métodos listados pelo ponto

de corte `move`. O bloco de código do *advice* consiste em uma única linha de código, que envia a mensagem "`Elemento movido`" para o console de erros.

Quando o código dos aspectos é compilado, o compilador examina o código do programa e procura os pontos identificados no ponto de corte. O código original do sistema é alterado, e o código dos *advices* é injetado em todos os pontos de junção que foram identificados.

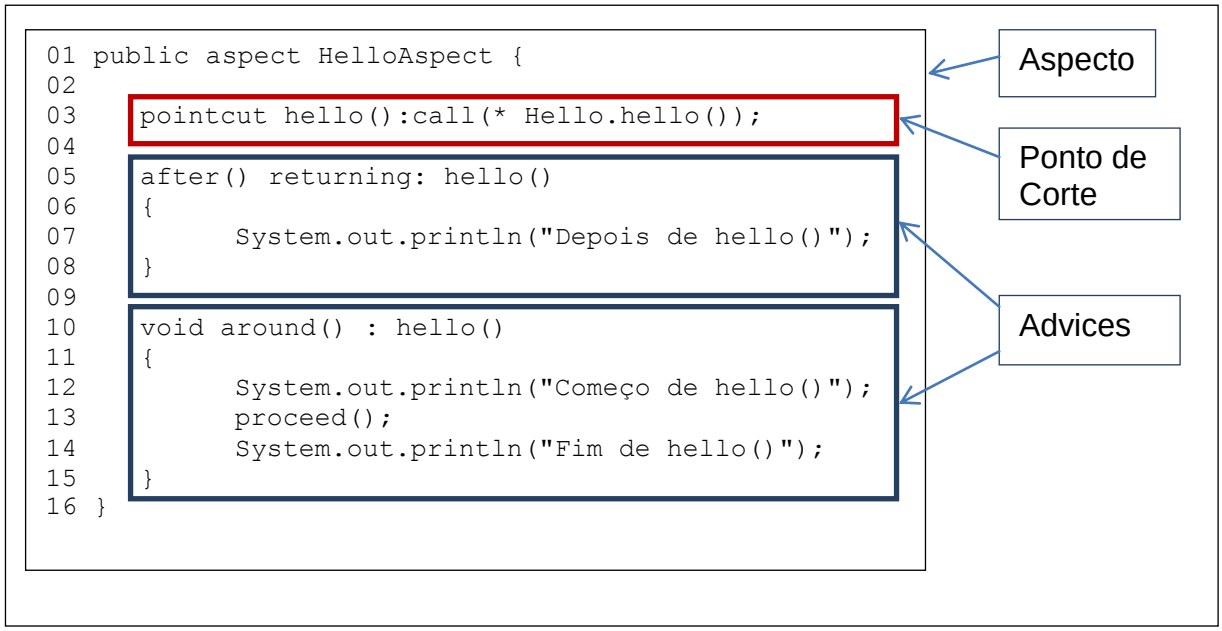
2.3.2. AspectJ

Há várias linguagens e *frameworks* de POA disponíveis, em diversas plataformas, mas a implementação mais popular atualmente é a linguagem AspectJ, feita para a plataforma Java. Os conceitos básicos de POA, implementados em AspectJ, são pontos de junção (*join points*), pontos de corte (*pointcuts*), *advices*, declarações inter-tipos (*inter-type declarations*), aspectos (*aspects*) (GRADECKI, 2003 e KICZALES et al., 2001).

- Pontos de junção (*join points*): são pontos específicos na execução de um programa onde o código dos aspectos pode ser injetado. Por exemplo, antes ou depois da chamada ao método `Classe1.metodo2()`. Os pontos de junção podem estar nas entradas e saídas de métodos, no tratamento de exceções, acessos a atributos de objetos e construtores.
- Pontos de corte (*pointcuts*): são expressões que definem um conjunto de pontos de junção. Por exemplo, antes da execução de qualquer método cuja assinatura siga o padrão "`*.get*()`", isto é, antes de todos os métodos cujo nome comece com "get", em todas as classes. São declarados com a palavra chave **`pointcut`**.
- *Advice*: é o trecho de código que deve ser injetado em todos os pontos de junção definidos por um ponto de corte. Por exemplo, pode ser uma chamada a um método de auditoria. São declarados com as palavras-chave **`before`**, **`after`** ou **`around`**, dependendo do instante em que são disparados.

- Declaração inter-tipos (*inter-type declaration*): é uma forma de adicionar atributos (variáveis) ou métodos (funções) a interfaces, classes e aspectos do sistema.
- Aspecto (*aspect*): agrupa pontos de corte, *advices* e declarações. É a unidade modular básica da linguagem. Um aspecto pode definir que um trecho de código (por exemplo, um *advice* de chamada a função de auditoria) deve ser executado para certo ponto de junção (por exemplo, "*.get*()", antes de qualquer métodos que comece com "get"). São declarados com a palavra-chave **`aspect`**.

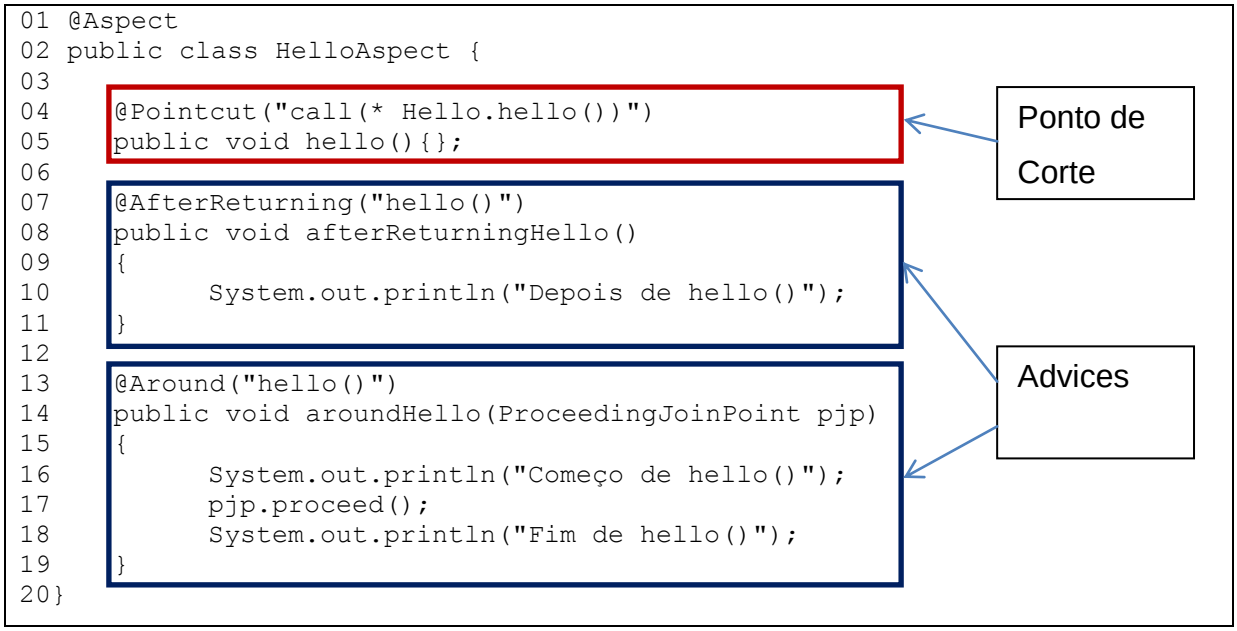
O código AspectJ pode ser escrito com duas sintaxes diferentes. A primeira é a sintaxe clássica declarativa, exemplificada na listagem 3. Os aspectos escritos nessa sintaxe são gravados em arquivos com a extensão "aj".



Listagem 3 - Exemplo de código AspectJ

Outra sintaxe é com o uso de anotações Java. Essa sintaxe, chamada @AspectJ, foi introduzida na versão 1.5 da linguagem, como resultado da união dos *frameworks* AspectJ e AspectWerkz. A listagem 4 mostra o mesmo exemplo de código reescrito com anotações.

A notação @AspectJ permite que a declaração de aspectos seja feita na forma de anotações Java, ou seja, na forma de metadados. Esta forma de programação é mais limitada que a forma declarativa, mas permite que o código seja compilado somente com o compilador Java, e sem necessidade de um compilador AspectJ.



Listagem 4 - Exemplo de código @AspectJ

A injeção do código dos aspectos é feita pelo combinador de aspectos (*aspect weaver*). Ela pode ser feita em três momentos, dependendo da forma como o compilador foi utilizado.

- Durante a compilação (*compile-time weaving*): Neste caso, há um pré-compilador que examina o código fonte e faz as alterações nos pontos apropriados.
- Depois da compilação (*post-compile weaving* ou *binary weaving*). O compilador da linguagem orientada a aspectos examina os binários compilados (*bytecode* no caso de Java) e altera esses binários injetando os binários dos *advices*.

- Em tempo de execução (*load-time weaving*). Quando as definições das classes são carregadas pela aplicação, o binário dos métodos é alterado em memória, antes de eles serem executados pela primeira vez.

Mais detalhes sobre a linguagem AspectJ e os combinadores de aspectos podem ser encontrados no site da fundação Eclipse: http:\\www.eclipse.org\aspectj

2.4. Personalização e POA

A programação orientada a aspectos pode ser vista como uma técnica para modularizar sistemas, isolando as responsabilidades transversais e com isso reduzindo o **emaranhamento** e o **espalhamento** de código (KICZALES, 1997). Há outras formas de se fazer isso como, por exemplo, utilizando-se técnicas e padrões de projeto OO. Para diferenciar as linguagens de programação orientadas a aspectos das demais abordagens, Filman e Friedman (2000) propuseram duas propriedades específicas, **quantificação** e **transparência** (*quantification and obliviousness*) que seriam indicadores de uma linguagem OA.

- Quantificação: uma linguagem OA deve permitir a criação de declarações quantificadas sobre o comportamento dos programas. Por exemplo, declarações na forma “No programa P, sempre que a condição C acontecer, execute a ação A”. No caso de AspectJ, a condição C é descrita pelo ponto de corte, e a ação A pelo *advice*.
- Transparência: as declarações quantificadas são feitas sobre programas escritos por programadores que ignoram a existência dessas declarações. Isto significa que o código da aplicação não precisa estar preparado para receber os aspectos. Esta é uma diferença de POA com alguns sistemas orientados a eventos, nos quais o programador precisa programar explicitamente as chamadas a eventos dentro do código.

Do ponto de vista da personalização de *software*, estas duas propriedades podem ser úteis.

- Quantificação: uma linguagem POA permite que o customizador defina, de forma centralizada, quais ajustes serão incorporados ao produto e em que condições eles se aplicam. No caso particular da linguagem AspectJ, ela permite que um trecho de código seja aplicado em vários pontos simultaneamente, desde que esses pontos satisfaçam a certas condições.
- Transparência: pode ser impraticável tentar prever de antemão todos os pontos do código que precisarão ser personalizados. Aspectos podem ser aplicados a qualquer parte do código, sem preparação prévia. Se um requisito de personalização surgir em um cliente em última hora, é possível adicionar este ajuste como um aspecto sem necessidade de reestruturação ou reescrita do código original.

A possibilidade que a POA oferece de adicionar código a um *software* mesmo sem ter acesso ao seu código fonte original torna-a uma ferramenta promissora para auxiliar a personalização de produtos. Ao mesmo tempo, a natureza invasiva dessa injeção de código pode afetar a estabilidade do produto, já que em princípio qualquer classe ou método do produto pode ser alterado. Além disso, os pontos de corte dos aspectos dependem da assinatura dos métodos que serão alterados. Consequentemente, as mudanças no código decorrentes da evolução natural do produto podem fazer com que assinatura dos métodos mude, e com isso os aspectos criados para uma versão do software deixem de funcionar na versão seguinte. Esse problema é chamado de *Pointcut Fragility* ou fragilidade dos pontos de corte (KOPPEN, 2004).

Essas desvantagens de POA podem ser mitigadas com a adoção de **Interfaces Orientadas a Aspectos**, descritas a seguir.

2.4.1. *Crosscutting Programming Interfaces* (XPI)

Griswald et al. (2006) e Sullivan (2010) propuseram uma solução para a fragilidade do ponto de corte, usando *Crosscutting Programming Interfaces*, ou XPI. XPI são **interfaces abstratas** que desacoplam o código dos aspectos e o código base. Elas definem regras de projeto (*Design Rules*) que guiam a definição dos

pontos de corte, limitando os tipos de pontos de corte que podem ser definidos pelo desenvolvedor que escreve o código dos aspectos. Desta forma, somente aspectos que seguem as regras definidas pelo criador do produto podem ser aplicados.

XPIs definem um conjunto de um ou mais pontos de junção, cada um expresso como um ponto de corte. Os aspectos concretos simplesmente utilizam os pontos de corte definidos na XPI, sendo, deste modo, restringidos em seu alcance de injeção de código. XPIs também podem possuir **contratos**, que são restrições de pré-condições e pós-condições que os aspectos concretos devem ser forçados a atender. As listagens 5, 6 e 7, extraídas de (SULLIVAN,2010), mostram exemplos de uma XPI e de seu contrato, criados para um software gráfico.

```
public aspect XFigureElementChange {

       public pointcut joinpoint(FigureElement fe):
              target(fe) && (call(void FigureElement+.set*(..))
              || call(void FigureElement+.moveBy(..))
              || call(FigureElement+.new(..)));

       public pointcut topLevelJoinpoint(FigureElement fe):
              joinpoint(fe) && !cflowbelow(joinpoint(FigureElement));
       public pointcut staticscope(): within(FigureElement+);
       public pointcut staticmethodscope():
              withincode (void FigureElement+.set*(..))
              || withincode(void FigureElement+.moveBy(..))
              || withincode (FigureElement+.new(..));
  }
```

Listagem 5 - Exemplo de implementação de XPI em AspectJ

```
public aspect FigureElementChangeContract {

       declare error: (!XFigureElementChange.staticmethodscope() &&
              set(int FigureElement+.*))
              : "Violação de contrato: só deve mudar o valor de um campo "+
              " FigureElement dentro de um método setter!";

       private pointcut advisingXPI(): adviceexecution();

       before(): cflow(advisingXPI())
              &&XFigureElementChange.joinpoint(FigureElement){
              ErrorHandling.signalFatal("Violação de contrato: o advice de"
              + " XFigureElementChange não pode mudar instâncias de FigureElement");
       }
}
```

Listagem 6 - Exemplo de contrato de XPI em AspectJ

```
public aspect DisplayUpdate {
       after(FigureElement fe): XFigureElementChange.topLevelJoinpont(fe) {
              Display.update(fe);
       }
}
```

Listagem 7 - Exemplo de aspecto usando uma XPI

XPI pode ser vista como uma camada intermediária entre os aspectos que de fato codificam as responsabilidades transversais e o código que recebe os aspectos. Ela tem função tripla: (1) desacoplar os aspectos concretos e o código do *kernel*, (2) controlar a aplicação dos aspectos concretos, por restrição dos pontos de corte onde os aspectos podem ser injetados e pelas limitações impostas pelos contratos, e (3) oferecer uma interface simplificada para quem for programar os aspectos concretos, escondendo a complexidade do *framework*. A diferença entre API e XPI é que XPIs abstraem responsabilidades transversais, ao contrário das APIs que abstraem somente métodos ou classes localizados.

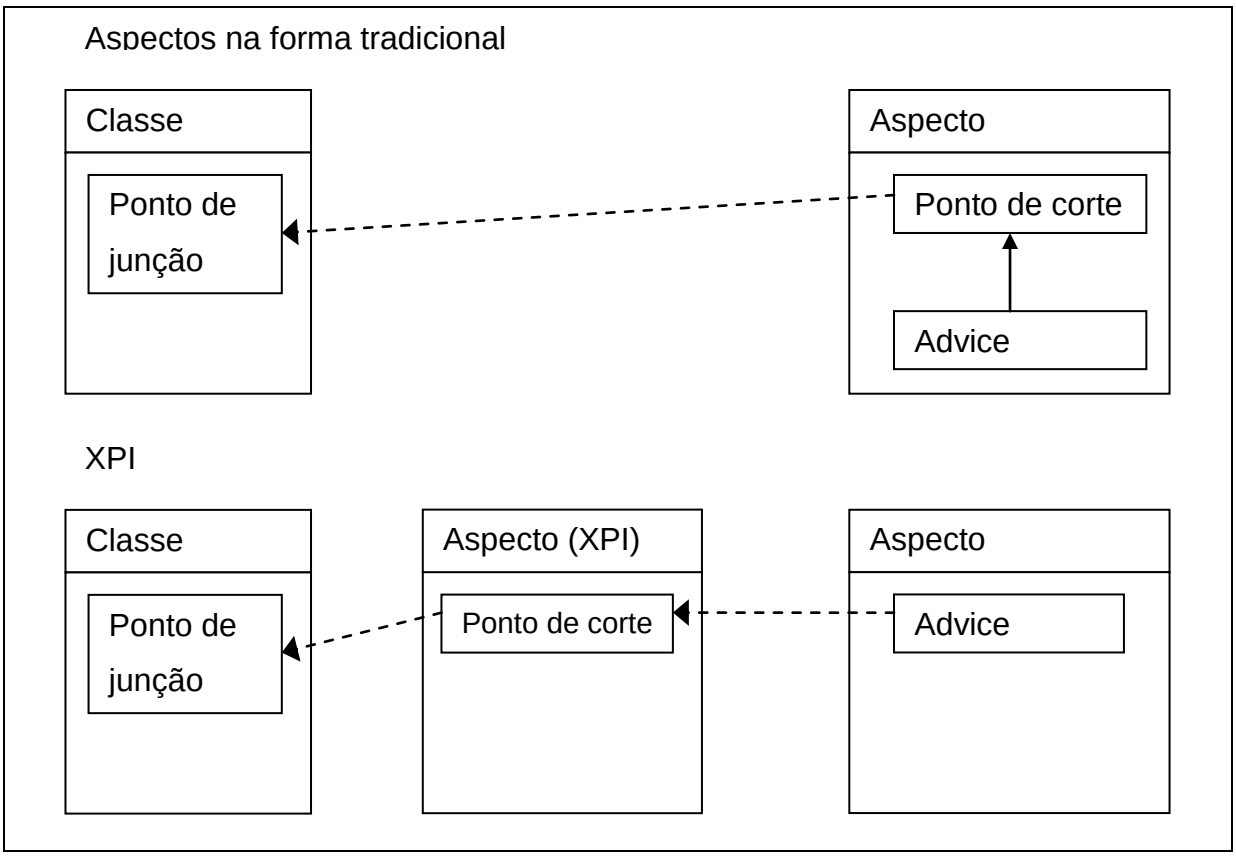


Figura 4 - Comparação entre aspectos com e sem XPI

Em resumo, XPI é uma forma de definir **pontos de extensão** na forma de pontos de corte. Com XPI, o implantador perde a liberdade de adaptar qualquer parte do produto via aspectos, e o arquiteto obtém maior controle sobre como o

código de personalização é aplicado ao produto. Se as XPIs se mantiverem estáveis a cada nova versão do produto, o código personalizável continua funcionando como esperado.

Porém, o uso de XPI como estratégia para controle de personalização de código apresenta duas limitações. Em primeiro lugar, XPI é só um conjunto de pontos de corte em AspectJ. Espera-se que o implantador siga a convenção de só definir aspectos associados à XPI, mas não há nenhum dispositivo no código ou na linguagem que impeça o implantador de criar aspectos que não usem a XPI e apontem diretamente para outras partes do código do produto. Se o objetivo for liberar acesso para que qualquer implantador possa personalizar o produto, mesmo que de outra empresa ou outra equipe, o ideal é que haja alguma forma de restringir o acesso, sem depender da disciplina dos programadores em seguir a convenção.

A segunda deficiência da XPI é que os contratos só se aplicam ao código criado pelos implantadores. Não há nenhum mecanismo na XPI ou na linguagem AspectJ que garanta o funcionamento correto da XPI se o código do produto for alterado. Isto exige maior disciplina da equipe que desenvolve o produto para manter as XPIs atualizadas e compatíveis com o código base a cada versão.

2.4.2. *Extension Join Points* (EJP)

Baseado na definição da XPI, Kulesza (2006) propõe o conceito de *Extension Join Points* (EJP) como uma forma de estruturar e documentar os pontos de extensão de um *framework*. A intenção é disciplinar o uso dos pontos de junção, limitando a aplicação dos aspectos a um conjunto restrito de pontos. Para isso, ele propõe um *framework* cuja arquitetura é composta de 4 elementos principais:

a) Núcleo do *framework* – implementa a funcionalidade obrigatória do produto. É similar a um *framework* OO tradicional. Sua estrutura contém *frozen-spots*, que representam funcionalidades padrão do produto, e classes *hot-spot*, que representam as variabilidades não transversais (ou seja, pontos de extensão).

b) Aspectos no núcleo – aspectos usados internamente no núcleo do *framework*, para uso exclusivo do fabricante do produto.

c) Aspectos de variabilidade – implementam funcionalidades opcionais ou alternativas. Esses elementos estendem os EJPs do *framework* com comportamentos transversais adicionais (*additional crosscutting behavior*).

d) Aspectos de Integração – definem aspectos que envolvem a integração de outras APIs ou *frameworks* com o produto. Esses aspectos também são extensões dos EJPs.

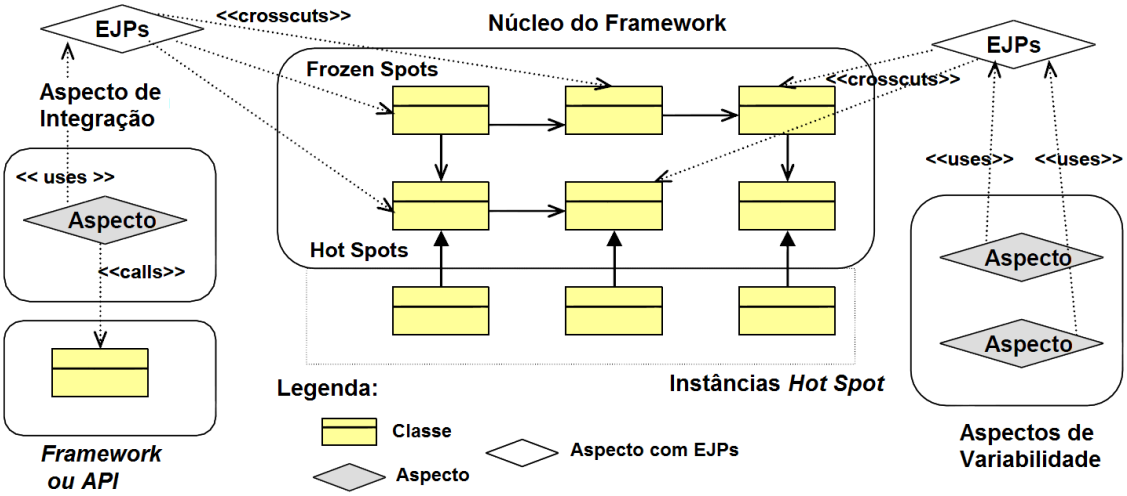


Figura 5 - EJPs aplicados a um *framework* (KULESZA, 2006)

Os EJPs são codificados em AspectJ de forma semelhante às XPIs. A diferença é a forma como os aspectos e contratos são classificados. Como já citado, os aspectos são separados em dois tipos: de variabilidade e de integração. Os contratos são separados em duas categorias, com subcategorias:

Contratos Internos: definem restrições para assegurar que refatorações e evolução do *framework* não afetem a funcionalidade dos aspectos de extensão. Há dois subtipos de contratos internos:

- Estruturais: garantem que as interfaces definidas pelos EJPs sejam implementadas no *framework*.
- Comportamentais: garantem que os EJPs afetem somente os eventos ou estados do *framework* que se quer expor.

Contratos de Extensão: asseguram que as restrições e invariantes do *framework* serão respeitadas. São classificados em:

- Estruturais: garantem que os aspectos concretos só podem estender os pontos de junção expostos pelos EJPs.
- Comportamentais: especificam os métodos e classes que poderão ser invocados pelos aspectos concretos.
- Invariantes: definem pré e pós-condições que devem ser preservadas antes e depois da execução dos *advices* dos aspectos concretos.

As figuras 8 e 9, extraídas de (KULESZA,2006) mostram um exemplo de implementação de EJP em AspectJ.

```
public aspect TestExecutionEvents {

	//Exigidos por: extensão RepeatAllTests
	public pointcut testExecution(Test test):
		target(test) && call (void Test.run(TestResult));
	// Exigidos por: extensão ActiveTestSuite
	public pointcut testSuiteExecution (TestSuite ts,TestResult rs):
		target(ts) && call (void TestSuite.run(TestResult)) && args(rs);
	// Exigidos por: extensão ActiveTestSuite
	public pointcut testExecutionFromSuite(TestSuite ts,Test t,TestResult rs):
		target(ts) && call (void TestSuite.runTest(Test, TestResult)) &&
		args(test, result);
	//Ainda não é usado por nenhuma extensão
	public pointcut testCaseExecution (TestCase tc, TestResult rs):
		target(tc) && call (void TestCase.run(TestResult)) && args(rs);
	//MÉTODOS AUXILIARES:
	protected pointcut EJPMethodsScope():
		withincode (void TestSuite.runTest(Test, TestResult)) ||
		withincode (void TestCase.runTest()) ||
		withincode (void TestSuite.run(TestResult));
		withincode (void Test.run(TestResult));
	// Escopo
	protected pointcut FWScope(): within(junit..*);
	//O propósito deste EJP é expor todos os pontos do framework
	// que resultam na execução de um teste
	protected pointcut MainPurpose(): call (void TestResult.run(Test));
}
```

Listagem 8 - Código AspectJ de um EJP feito para o framework JUnit

```
public aspect TestExecutionEventsContracts {
	//Contrato comportamental interno
	declare error:
		(!TestExecutionEvents.EJPMethodsScope() &&
		TestExecutionEvents.MainPurpose() ):
		"Violação de contrato: Execução de testes deve ocorrem "+
		"por meio de um desses métodos: Test.run(), TestSuite.run(),"+
		"TestSuite.runTest(),TestCase.run(), TestCase.runTest()";
	//Contrato comportamental da extensão
```

```
        public pointcut variabilityaspects(): within(variabilityaspects..*);
        before() : cflow ( adviceexecution() && !variabilityaspects() ) &&
                ( call(* *(..)) && TestExecutionEvents.FWScope() ){
                throw new RuntimeException("Violação de contrato: nenhum aspecto" +
                " exceto aspectos de variabilidade, podem acessar elementos do "
                " framework JUnit.");
        }
  ...
}
```

Listagem 9 - Exemplo de Contrato do EJP

Em resumo, EJPs podem ser vistos como uma forma estruturada de definição de XPIs, com o propósito de estender um *framework* através de aspectos. XPIs e EJPs podem ser usados para aumentar o controle do arquiteto do produto sobre os ajustes de personalização, melhorando também a resiliência dos ajustes à evolução do produto. Mas os EJPs, por serem baseados em XPI, sofrem as mesmas limitações desta, que são a falta de um contrato entre EJP e código do *kernel* e falta de um mecanismo para impedir a criação de aspectos que contornem os EJPs.

2.4.3. *Join Point Interfaces* (JPI)

Inostroza et al. (2011) introduziram o conceito de *Join Point Interfaces* (JPI), ou Interfaces de Ponto de Junção, para tratar o problema da fragilidade dos pontos de corte. Trata-se de uma extensão da linguagem AspectJ que define uma interface entre aspectos e código base.

A listagem 10, adaptada de (INOSTROZA, 2011), mostra como definir e usar uma JPI. Trata-se de um exemplo de código de um carrinho de compras, e de um aspecto que aplica descontos de 5% para clientes que fazem aniversário.

A declaração da JPI é feita na linha 1, com a palavra chave ***jpi***. Ela é idêntica a uma assinatura de método, com nome, parâmetros, tipo de retorno e opcionalmente a cláusula *throws* para lançamento de exceções. No exemplo, a JPI *checkingOut* declara uma interface para o evento de adição de compras em um carrinho.

```
01 jpi void checkingOut(Item item, float price, int amount, Customer cus);
02
03 class ShoppingSession {
04     exhibit void checkingOut (Item i, float price, int amount, Customer c):
05         execution (* checkOut (..) ) && args (i, price, amount, c) ;
06
07     ShoppingCart sc = new ShoppingCart() ;
08     Invoice inv = new Invoice() ;
09
10     void checkOut (Item item , float price, int amount, Customer cus) {
11         sc.add (item, amount);
12         inv.add (item, amount, cus);
13     }
14 }
15
16 aspect Discount {
17     void around checkingOut (Item item, float price, int amt , Customer cus ) {
18         int factor = cus.hasBirthday() ? 0.95 : 1;
19         proceed(item , price*factor, amt, cus) ;
20     }
21 }
```

Listagem 10 - Exemplo de JPI (Interface de Ponto de Junção)

A classe *ShoppingSession* é uma classe do código base. O trecho das linhas 7 a 13 é código na linguagem Java padrão. A declaração ***exhibits*** na linha 4 liga a JPI aos pontos de junção dentro da classe, usando uma declaração de ponto de corte em AspectJ. No exemplo, o ponto de junção da jpi *checkingOut* é a execução do método *checkOut*.

E por fim, o aspecto *Discount* contém um *advice* à jpi *checkingOut* que aplica o desconto desejado.

A forma como as JPIs são definidas difere do uso padrão de POA em três pontos importantes.

- Os pontos de junção são definidos explicitamente como interfaces, as JPIs. A JPI é a interface entre os aspectos e o código base. Os aspectos nunca são aplicados diretamente ao código base, eles devem obrigatoriamente passar pela JPI. E as classes do código base devem declarar explicitamente quais JPIs elas expõem. Caso haja mudanças na definição de uma JPI, o compilador checa se o código base e os aspectos continuam compatíveis.
- A definição dos pontos de corte é feita no código base, na declaração *exhibits*. Cada classe é responsável por declarar quais JPIs ela suporta, e como esta JPI é implementada. Desta forma, o programador que codifica a classe tem total controle sobre como ela receberá os *advices* dos aspectos.

- O aspecto não contém nenhuma definição de pontos de corte. Ele fica livre de qualquer referência a elementos do código base, o que o torna mais reutilizável.

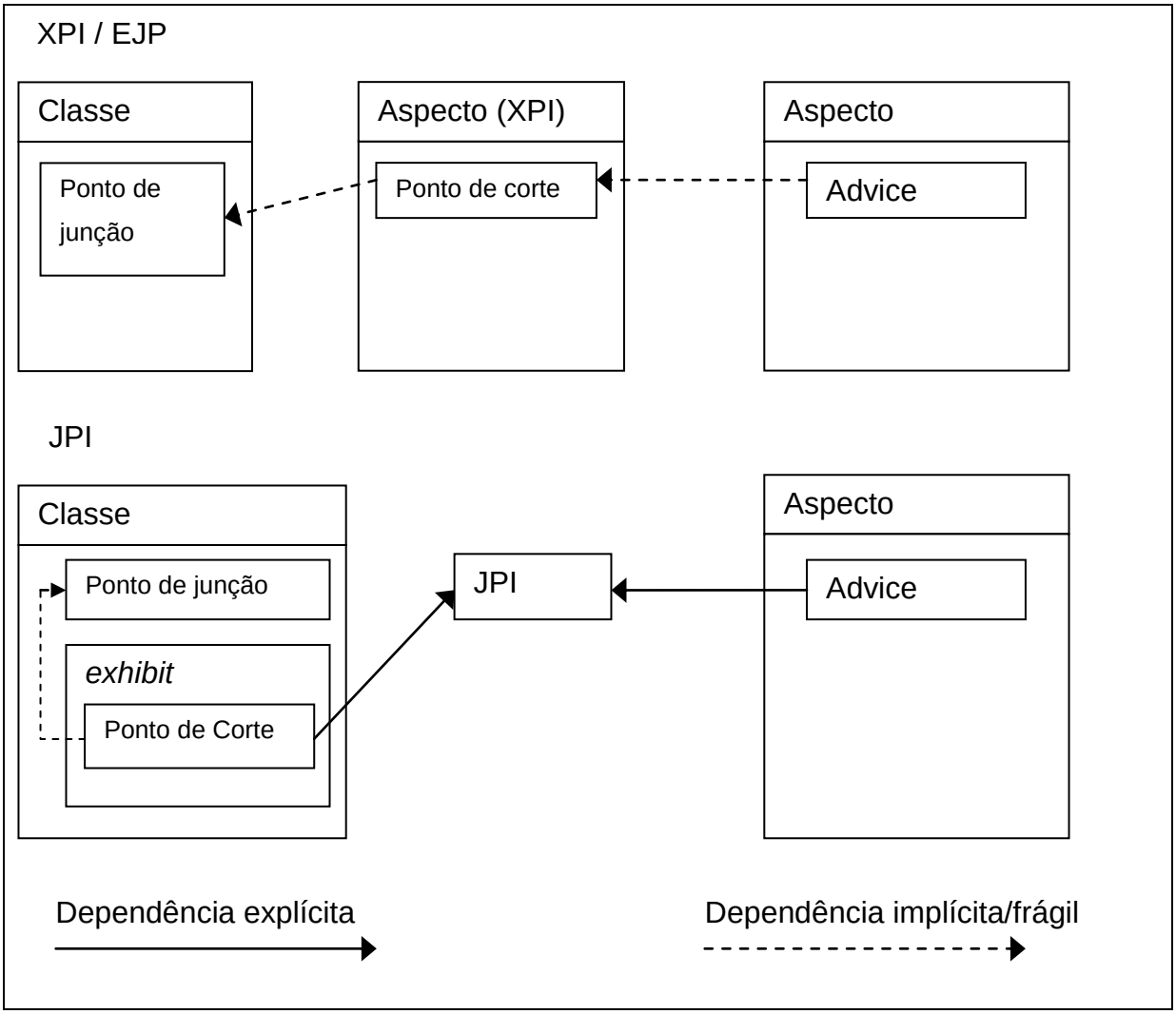


Figura 6 - Comparação entre JPI e XPI.

A ideia de definição de interfaces da JPI é semelhante à da XPI e do EJP. Mas traz a vantagem de não depender de uma convenção de programação. A definição e a aplicação das interfaces são feitas com novos elementos da linguagem de programação, o que garante a validação das interfaces durante a compilação. A fragilidade dos pontos de corte ainda persiste, porque os pontos de corte definidos dentro de cada classe ainda podem deixar de funcionar caso a assinatura de algum

método mude. Mas ela fica mais localizada, já que os pontos de corte se aplicam somente à classe à qual pertencem.

Do ponto de vista da personalização de software, JPIs podem ser encaradas como definições de pontos de extensão. Os aspectos contêm o código personalizado, e cada classe do código base seria responsável por definir quais pontos de extensão se aplicam a elas e como o código personalizado será tratado internamente. JPI servem de fachada e oferecem uma interface simplificada para programador que personaliza o produto.

2.4.4. *Join Point Types* e *Polymorphic Pointcuts*

Steimann (2009) propôs uma extensão das linguagens Java e AspectJ para incorporar os conceitos de Tipos de Ponto de Junção (*Join Point Types*) e Pontos de Corte Polimórficos (*Polymorphic Pointcuts*). A listagem 11 mostra um exemplo de código, também de um carrinho de compras.

A proposta é semelhante à JPI de Inostroza, mas com uma implementação um pouco diferente. A ideia é interpretar a execução dos pontos de junção como eventos. Os pontos de junção possuem tipos que são declarados de forma semelhante a classes. Cada vez que um ponto de junção é executado no programa, uma nova instância do tipo correspondente de ponto de junção é criada na memória. Este tipo contém atributos que serão acessados e possivelmente alterados pelos métodos dos aspectos. As linhas de 1 a 4 mostram a declaração de um `joinpointtype`.

Os pontos de corte são definidos dentro das classes que possuem os pontos de junção. Os pontos de corte são considerados polimórficos pelo fato de cada classe poder definir expressões diferentes de pontos de corte diferentes para o mesmo tipo de ponto de junção. Nas linhas 7 e 15 há duas declarações distintas de pontos de corte, em classes separadas. A cláusula `exhibits` é usada nas classes para indicar quais tipos de pontos de junção elas expõem.

Os aspectos usam a cláusula `advises` para indicar os tipos de pontos de junção aos quais eles se aplicam. Assim como na JPI, o aspecto não contém

expressões de pontos de corte, não havendo dependência entre aspectos e detalhes de implementação do código base.

Uma das vantagens dessa proposta sobre a JPI de Inostroza é definir os aspectos e os tipos de pontos de junção de uma forma muito semelhante a de classes comuns. É possível definir relações de herança para aspectos de pontos de junção. Isto torna a programação dos aspectos muito similar à POO tradicional. A única grande diferença em relação à POO é a inclusão de expressões de pontos de corte dentro das classes do código base.

```
1 joinpointtype Buying {
 2       Item item;
 3       int amount;
 4 }
 5
 6 class ShoppingCart exhibits Buying {
 7       pointcut Buying : execution(* add(Item, int)) && args(item, amount);
 8
 9       void add(Item it, int amount) {
10                System.out.println("ShoppingCart.add("+it+","+amount+")");
11       }
12 }
13
14 class Log exhibits Buying {
15       pointcut Buying :
16            execution(* add(Item, int, ..)) && args(item, amount, ..);
17
18       ...
19       }
20
21 aspect ShoppingCart advises Buying {
22       void around (Buying jp) {
23           System.out.println("\t** BonusProgram: around  -> "+jp.getClass());
24           if (jp.item.category == Item.BOOK) {
25               System.out.println("\t[buy]\t-> Item == BOOK : add "+
26                                  jp.amount / 2+" to amount "+jp.amount);
27               jp.amount += jp.amount / 2;
28           }
29           proceed(jp);
30       }
31
32 }
```

Listagem 11 - Tipos de ponto de junção e pontos de corte polimórficos

2.4.5. Eventos Declarativos Orientados a Objetos

Gasiunas (2011) argumenta que os pontos de junção da POA podem ser considerados eventos implícitos, isto é, eventos que existem no programa de forma não declarada. Os pontos de corte podem ser vistos como definições **declarativas**

desses eventos. Esses eventos declarativos se contrapõem aos eventos **imperativos** normalmente utilizados em POO, que são codificados em Java seguindo o padrão de projeto *Observer* ou com *delegates* na linguagem C#.

Gasiunas propõe incorporar o conceito de “Eventos Declarativos” às linguagens de programação orientada a objetos. Como prova de conceito foi criada a linguagem EScala, um extensão da linguagem Scala com suporte a eventos. Ela incorpora a noção de eventos como membros de objetos, como em C#, mas com a possibilidade de defini-los com expressões declarativas ao estilo de POA.

A listagem 12 mostra um exemplo de eventos definidos em EScala.

```
1 abstract class Figure {
 2   ...
 3   event moved[Unit] = after(moveBy)
 4   abstract event resized[Unit]
 5   event changed[Unit] = resized || moved || after(setColor)
 6   event invalidated[Rectangle] = changed.map(_ => getBounds())
 7   ...
 8   def moveBy(dx: Int, dy: Int) {
 9     position.move(dx, dy)
10   }
11   def setColor(col: Color) {
12     color = col
13   }
14   def getBounds(): Rectangle
15   ...
16 }
17
18 class Connector(val start: Figure, val end: Figure) {
19   start.changed += updateStart
20   end.changed += updateEnd
21   ...
22   def updateStart() { ... }
23   def updateEnd() { ... }
24   ...
25   def dispose() {
26     start.changed -= updateStart
27     end.changed -= updateEnd
28   }
29 }
```

Listagem 12 - Eventos Declarativos em EScala (GASIUNAS, 2010)

O exemplo mostra o código de classes de figuras usadas em um editor gráfico. A classe abstrata *Figure* representa uma figura geométrica, e a classe *Connector* uma conexão entre figuras. Os eventos *moved*, *changed* e *invalidated* (linhas 3, 5 e 6) são definidos declarativamente. O evento *moved* é definido pela expressão *after(moveBy)*, indicando que ele é disparado logo após a execução do método *moveBy*. O evento *changed* é definido pela união dos eventos *resized*,

moved e o ponto após a execução do método *setColor*. O evento invalidated dispara após o evento *changed*, e o operador *map* na linha 6 é uma *closure* que chama o método *getBounds*. A classe Connector (linhas 18 a 29) registra os métodos *updateStart* e *updateEnd* como reações aos eventos *changed* disparados pelas figuras *start* e *end*.

A definição dos três eventos *moved*, *changed* e *invalidated* é feita com uma sintaxe diferente daquela usada nos pontos de corte de AspectJ, mas o funcionamento é semelhante. Dessa forma, o uso de aspectos é substituído por eventos, e os aspectos deixam de ser usados.

Essa abordagem simplifica a programação ao dispensar uma linguagem orientada a aspectos, o que pode ser atraente como forma de facilitar o processo de personalização de *software*. Por outro lado, ao dispensar totalmente POA, perde-se o poder das expressões de pontos de corte.

2.4.6. *Closure Joinpoints*

Bodden (2011) propõe estender a JPI de Inostroza et al. (2011) para incluir o conceito de Pontos de Junção de Blocos. O objetivo é permitir que os aspectos sejam aplicados também a blocos de código, isto é, trechos de código dentro dos métodos. Isto é feito marcando-se um bloco com a palavra-chave `exhibit`, seguida da definição do ponto de junção exposto com seus parâmetros. Um exemplo é mostrado na listagem 13.

```
 1 import static BonusProgram.Buying;
 2 class ShoppingSession {
 3   int totalAmount = 0;
 4   ShoppingCart sc = new ShoppingCart();
 5
 6   void buy(final Item item, int amount) {
 7     Category category = Database.categoryOf(item);
 8
 9     //changes start here
10     totalAmount = exhibit Buying(Category c, int amount) {
11       sc.add(item, amount);
12       return totalAmount + amount;
13     }(category,amount);
14   }
15 }
16
17 aspect BonusProgram {
18   joinpoint int Buying(Category cat, int amount);
19   int around Buying(Category cat, int amt) {
20     if (cat==Item.BOOK)
21     amt += amt / 2;
```

```
22      return proceed(cat, amt);
23    }
24 }
```

Listagem 13 - Exemplo de Closure Joinpoint (BODDEN, 2011)

O bloco entre as linhas 10 e 13 expõe uma JPI tipo `Buying`, que recebe como parâmetro as variáveis `category` e `amount`. O aspecto é definido nas linhas 17 a 24.

Algumas das vantagens desta proposta:

- Em certos casos, pode melhorar a legibilidade do código. A JPI em sua forma original só permite a definição de pontos de corte para métodos. Se fosse usada somente a JPI original, seria necessário extrair o trecho de código a ser exposto em um método separado, somente para associá-lo a um ponto de corte.
- Permite a escolha de somente algumas das variáveis do bloco para serem expostas na JPI. Por exemplo, na linha 11 a variável `item` é usada, mas esta variável é invisível para o aspecto `BonusProgram`. Somente as variáveis `category` e `amount` podem ser usadas no aspecto. Isto não seria permitido no caso da extração desse bloco em um método separado. Como o bloco inclui variáveis do escopo externo a ele, ele pode ser considerado um fechamento (*Closure*), isto é, uma função anônima que captura valores do escopo léxico externo a ela.

2.4.7. Outras soluções

Bagherzadeh (2011) define Contratos Translúcidos (*Translucid Contracts*) como uma forma de implementar XPIs e outras formas de contrato baseados em aspectos com a linguagem *Ptolemy*. Os objetivos são (1) limitar os pontos onde os aspectos serão injetados, e (2) permitir maior controle sobre o fluxo de execução do código dos aspectos. A técnica inclui a definição de eventos, mas de uma forma diferente daquela usada por (GASIUNAS 2010).

Bockisch (2011) também propõe uma abordagem baseada em eventos, mas ao contrário de (GASIUNAS 2011) propõe uma composição entre eventos e uma extensão da linguagem AspectJ.

Vaucouleur (2009) propõe o conceito de *Code Queries*, (consultas de código) como uma estratégia para a personalização de ERPs. A ideia é permitir que o implantador leia o código fonte do produto original e decida quais pontos ele quer alterar. Ele só tem permissão de leitura do código fonte, mas pode injetar o seu código personalizado através da definição de *code queries*. Uma *code query* é a descrição de um trecho de código que deverá ser substituído pelo código personalizado ou sofrer injeção de código extra.

Nas linguagens tradicionais de POA, como AspectJ, a injeção de código é feita com base na assinatura dos métodos e das classes. Com *code queries* a injeção é feita usando como referência o código dentro dos métodos e classes. Essa abordagem dá uma liberdade muito maior ao implantador, que pode ajustar cada pedaço de código às suas necessidades. Mas o arquiteto do produto perde controle sobre o que o implantador faz. Isto aumenta a chance de os ajustes de personalização desestabilizarem o sistema. Além disso, torna os ajustes mais suscetíveis a quebra de compatibilidade com a evolução do produto, já que eles podem ficar muito mais atrelados a detalhes de implementação.

Code queries são um passo na direção contrária a XPI ou JPI. Estas tentam restringir o acesso do implantador somente a alguns pontos de extensão. *Code queries* abrem o acesso a qualquer parte do código.

2.5. Resumo

Customização de *Software* pode ser feita de forma interativa (configuração, *scripts*, editores visuais) ou por adaptação no código fonte. Quando é feita por adaptação de código fonte, vários agentes podem estar envolvidos (fabricante do produto, implantador/customizador ou cliente/usuário final), em várias fases do ciclo do produto (desenvolvimento, refinamento, implantação e uso). Entre as técnicas e arquiteturas usadas para facilitar personalização do código fonte, podemos citar:

- Técnicas baseadas em POO: herança, interfaces, parapolimorfismo, métodos parciais, *mixins*, *traits*.
- Eventos e callbacks
- Arquitetura em camadas
- Linhas de produtos de software
- Padrões de Projeto, como *Strategy, Template Method, Adapter, Decorator* e *Abstract Factory*.
- Técnicas baseadas em Programação Orientada a Aspectos (POA)

A programação orientada a aspectos, criada para codificar responsabilidades transversais, a princípio parece ser uma boa alternativa para criação de código de personalização. Ela oferece a possibilidade de injetar código personalizado no código base, alterando o comportamento do *software* para que atenda as necessidades de um cliente ou usuário específico. Na prática, o uso de POA esbarra em duas dificuldades: a fragilidade dos pontos de corte e a perda de controle do arquiteto do produto sobre como o customizador vai alterar o *software*.

Uma solução é definir uma interface ou contrato entre os aspectos e o código base, de forma transferir a definição dos pontos de corte para as classes do código base. As soluções encontradas na literatura, ou dependem de convenções de programação (XPI e EJP), ou de extensões da linguagem de programação (JPI, *Closure Joinpoints*, *Join Point Types*, Eventos Declarativos).

Quando a definição dos pontos de corte é transferida para as interfaces ou contratos, tira-se do customizador a possibilidade de definir os pontos de corte, o que passa a ser responsabilidade do arquiteto do produto. Esta solução guarda alguns pontos em comum com a personalização feita por eventos ou *callbacks*. A principal diferença é que com o uso de aspectos, os eventos são definidos de forma declarativa.

3 PROPOSTA: O *FRAMEWORK* ACF

O capítulo 2 explorou o problema da customização de software através de programação orientada a aspectos, mostrando vantagens e limitações dessa abordagem. A literatura mais recente tem apontado o uso de interfaces ou contratos orientados a aspectos como uma possível alternativa para contornar algumas dessas limitações.

Neste capítulo será apresentado o ACF (*Aspect-based Customization Framework*), um *framework* de customização baseado em aspectos. Sua proposta é permitir a personalização de produtos de *software* por terceiros, mesmo em pontos onde o projeto do produto não previu a necessidade de customização.

As interfaces orientadas a aspectos são o elemento central da arquitetura do ACF. Sua função dentro do *framework* é desacoplar o código customizado do código do produto, permitindo que ambos evoluam de forma independente. No capítulo 2 foram citadas várias as propostas de definição de interfaces orientadas a aspectos. Dentre elas, optou-se pela adoção da JPI de (Inostroza et al. 2011) como base para a construção do *ACF*. JPIs se mostram vantajosas em relação a XPIs ou EJPs, porque resolvem de forma mais eficiente o problema da fragilidade dos pontos de corte.

3.1. @JPI – *Join Point Interfaces* através de anotações

Foi criada para este trabalho uma versão modificada da JPI, na forma de uma extensão da notação @AspectJ. Esta forma de declaração de JPIs foi chamada de **@JPI**. As razões para a escolha de uma solução baseada em anotações foram as seguintes:

a) O uso de anotações permite a programação de aspectos somente com a linguagem Java.

b) A notação @AspectJ é aceita por ferramentas populares, como o *Framework* Spring e o compilador AspectJ Eclipse.

c) A única implementação existente de um compilador JPI apresenta limitações. Ela se baseia no compilador AspectBench (ABC – *AspectBench Compiler*) criado por grupos da Universidade de Oxford e da Universidade McGill (Canadá). AspectBench é um compilador de código aberto da linguagem AspectJ, cujo principal propósito é permitir a criação de extensões e otimizações da linguagem. Por ser um compilador experimental, ele possui uma série de limitações que dificultam o seu uso no contexto de customização de *software*.

- O combinador de aspectos (*aspect weaver*) do AspectBench só permite injeção de código em tempo de compilação ou pós-compilação (*compile-time weaving* ou *post-compile weaving*). Outros combinadores, como o AspectJ da fundação Eclipse, permitem também a combinação de aspectos durante o carregamento do *software* (*load-time weaving*). A combinação durante o carregamento é mais apropriada para uso em customização de software, já que dispensa a recompilação do produto ou a alteração de seus binários.
- O AspectBench está defasado em relação aos avanços da linguagem AspectJ. A última versão deste compilador é de abril de 2008, e não incorpora as características da linguagem Java da versão 1.5 e 1.6. Não suporta, por exemplo, anotações, tipos genéricos. Também não aceita a notação @AspectJ.
- Não possui integração com os ambientes de desenvolvimento mais usados, como Eclipse ou NetBeans, nem com *frameworks* populares, como Spring e EJB.

A @JPI acrescenta duas novas anotações ao @AspectJ, `@Jpi` e `@Exhibit`, que têm as mesmas funções das palavras chave **jpi** e **exhibits** propostas por Inostroza. As figura 20 e 21 mostram a declaração de uma JPI com a linguagem original e com a notação @JPI.

A anotação `@Jpi` serve para declarar a JPI, como mostrado nas linhas 1 a 5 da listagem 15. Uma diferença importante em relação à linguagem original (linha 1 da listagem 14) é que na nova notação uma JPI é declarada como parte um aspecto, de forma semelhante a um ponto de corte. Uma das razões para isso é que a linguagem Java não permite a declaração de funções fora de classes. Outro motivo

é permitir que os *advices* apontem para a JPI da mesma forma como eles apontariam para um ponto de corte.

```
01 jpi void checkingOut(Item item, float price, int amount, Customer cus);
02
03 class ShoppingSession {
04     exhibits void checkingOut (Item i, float price, int amount, Customer c):
05         execution (* checkOut (..) ) && args (i, price, amount, c) ;
06
07     ShoppingCart sc = new ShoppingCart() ;
08     Invoice inv = new Invoice() ;
09
10     void checkOut (Item item , float price, int amount, Customer cus) {
11         sc.add (item, amount);
12         inv.add (item, amount, cus);
13     }
14 }
15
16 aspect Discount {
17     void around checkingOut (Item item, float price, int amt , Customer cus ) {
18         int factor = cus.hasBirthday() ? 0.95 : 1;
19         proceed(item , price*factor, amt, cus) ;
20     }
21 }
```

Listagem 14 - Exemplo de JPI

```
01 @Aspect
02 class JoinpointInterfaces {
03     @Jpi
04     public void checkingOut(Item item, float price, int amount, Customer cus){};
05 }
06
07 @Exhibits({
08 @JpiPointcut(jpi="JoinpointInterfaces.checkingOut(Item item, float price,
09                   int amount, Customer c)",
10              pointcut="execution(* checkOut (..) ) && args(item,price,amount,c)"
11 "})
12 class ShoppingSession {
13
14     ShoppingCart sc = new ShoppingCart() ;
15     Invoice inv = new Invoice() ;
16
17     void checkOut (Item item , float price, int amount, Customer cus) {
18         sc.add (item, amount);
19         inv.add (item, amount, cus);
20     }
21 }
22
23 @Aspect
24 class Discount {
25
26     @Around("JoinpointInterfaces.checkingOut(Item, float, int, Customer) &&
27              args(item, price, amount, c)")
28     public void discount(Item item, float price, int amount, Customer c,
29                          ProceedingJoinPoint thisJoinPoint ) {
30         int factor = cus.hasBirthday() ? 0.95 : 1;
31         thisJoinPoint.proceed(item , price*factor, amt, cus) ;
32     }
33 }
```

Listagem 15 - Exemplo de @JPI

O aspecto `Discount`, contido nas linhas 23 a 33, declara o *advice* `discount`, que é aplicado à JPI. A sintaxe usada na declaração @Around segue o padrão do AspectJ para pontos de corte. Não há necessidade de uma expressão diferente para JPIs, elas são vistas pelos *advices* como pontos de corte comuns.

As linhas 7 a 21 contém o código da classe `ShoppingSession`, que tem parte de sua funcionalidade exposta na JPI. A anotação `@Exhibits`, nas linhas 7 a 11, define um ponto de corte para a execução do método `checkOut` que será exibido pela JPI `JoinpointInterfaces.checkingOut`.

3.2. Requisitos para o *Framework*

Os requisitos que o ACF deve atender são descritos a seguir.

a) Customização com programação orientada a aspectos

A inclusão do código personalizado poderá ser feita através de aspectos.

b) Oferecer exclusivamente Interfaces Orientadas a Aspectos

Para que o fabricante do produto tenha controle sobre como a personalização é feita e para evitar o problema da fragilidade dos pontos de corte, os pontos de extensão (isto é, pontos de junção) devem ser definidos através de interfaces OA.

c) Deve permitir personalizações imprevistas

Em algumas situações, é possível que os usuários peçam alterações no produto que não foram previstas pelos os arquitetos do produto, exigindo alterações em funcionalidades para as quais não há pontos de extensão. Um dos requisitos do *framework* é o de facilitar a criação de novos pontos de extensão, e ao mesmo tempo evitar que isso cause alterações significativas na arquitetura do produto.

d) Deve permitir que o produto evolua sem afetar customizações já criadas

O código personalizado criado pelos implantadores deve continuar funcionando corretamente, mesmo que a evolução do produto exija alterações de partes do *software* que foram personalizadas.

e) Não pode depender de alteração no código fonte original

O *software* é entregue já compilado para os usuários finais. Não será permitida a alteração do código fonte para recompilação do produto.

3.3. Arquitetura do ACF

O *ACF* é composto de três partes, mostradas na figura 7.

a) Produto: é o *software* original, que receberá os ajustes de customização.

b) Interface de Customização: é a camada que expõe as funcionalidades do produto que poderão ser ajustadas.

c) Customização: é o código criado pelos implantadores. Sua utilidade é atender aos requisitos específicos de um cliente.

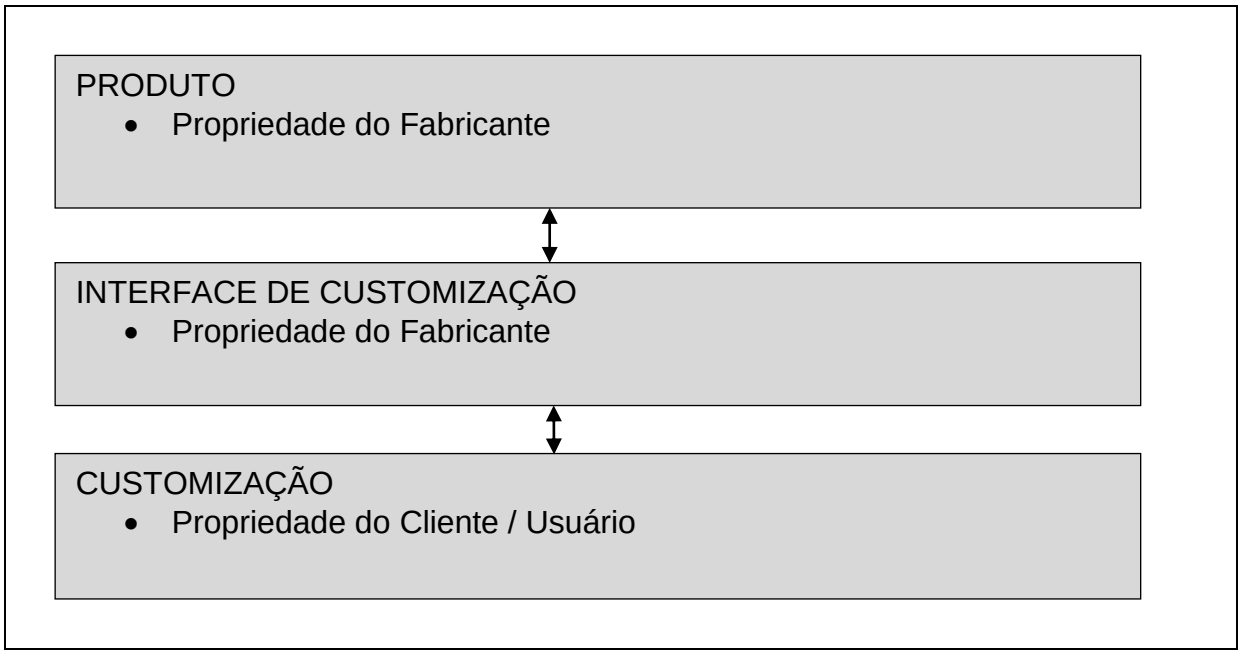


Figura 7 - Elementos do *Framework* de Customização

O “Produto” e a “Interface de Customização” são mantidos pelo fabricante do produto. O código da “Customização” é de propriedade dos clientes/usuários.

As camadas são criadas e gerenciadas por equipes com interesses conflitantes. O objetivo do fabricante do produto é fazê-lo evoluir para atender a todos os clientes e usuários da melhor forma possível, evitando adicionar

funcionalidades que não façam sentido para a maioria deles. Também procura manter a arquitetura simples e estável ao longo do tempo, e tem uma visão de longo prazo da evolução do produto.

Já os clientes e usuários, que são os donos da camada de customização, têm como principal preocupação resolver problemas bem específicos de sua organização. Normalmente querem soluções rápidas e simples. Têm pouca preocupação com a evolução do produto. O ideal para o cliente seria poder adaptar customizações diretamente em qualquer parte do produto, quando surgisse a necessidade.

A finalidade da camada de “Interface de customização” é tentar conciliar esses interesses conflitantes. Ela serve para isolar o produto das customizações, permitindo que o produto evolua sem que as customizações deixem de funcionar. O isolamento é obtido com o uso de JPIs, definidas na camada “Interface de Customização”.

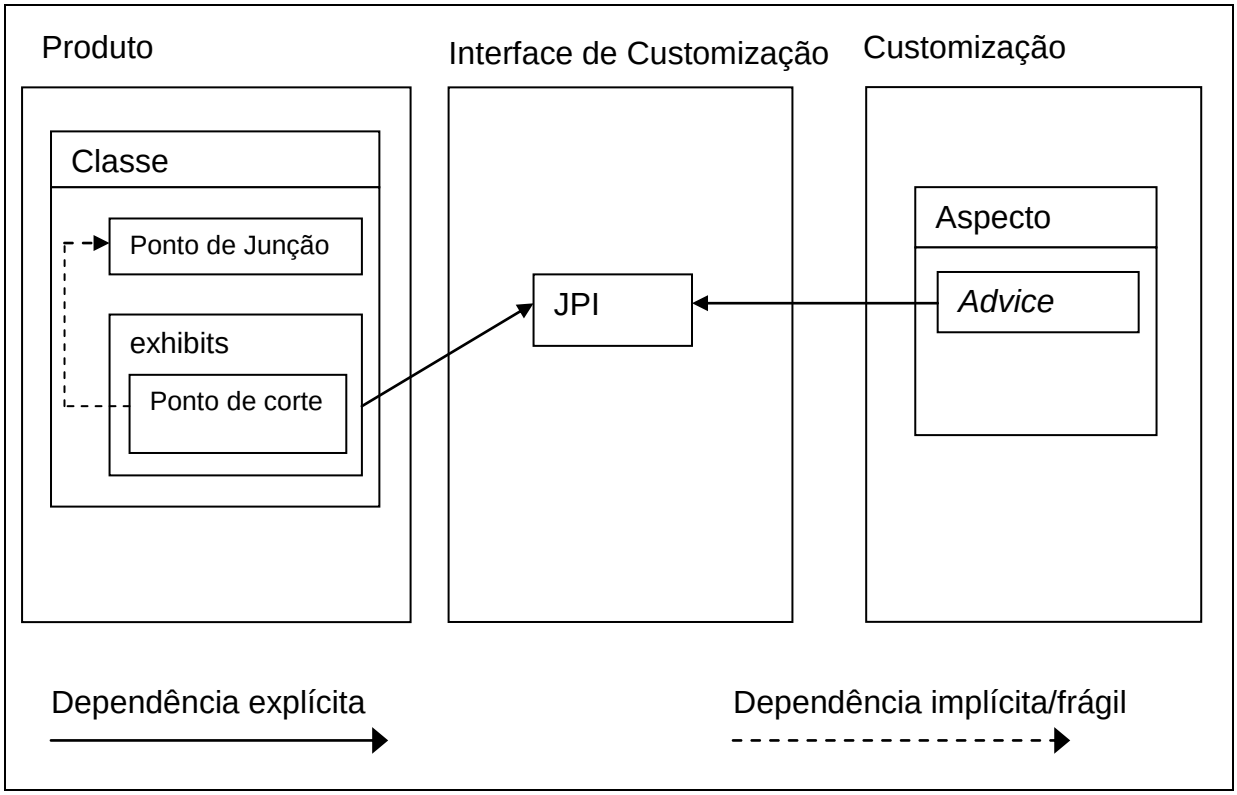


Figura 8 - Uso de JPI no ACF

As JPIs definem os contratos entre customizações e produto. Quando o fabricante do produto deseja adicionar um novo ponto de extensão, ele cria uma nova JPI para definir como ele será exposto para os clientes. Em seguida, são criados um ou mais pontos de corte dentro do produto que são associados a essa JPI por expressões *exhibits*. Quando algum cliente quiser injetar ajustes nesse ponto de extensão, basta criar um aspecto contendo somente um *advice*, e aplicá-lo à JPI correspondente. A figura 8 mostra esses elementos nas três camadas do ACF.

JPIs são contratos fixos, portanto não podem ser alteradas em versões futuras do produto. Se a evolução do *software* exigir que as classes que contêm pontos de extensão sejam reescritas, basta manter a JPI inalterada e garantir que os pontos de corte continuem compatíveis. Esta compatibilidade é checada pelo combinador de aspectos, que verifica a correção das anotações *exhibits* durante na compilação do produto. A compatibilidade entre customizações e JPI também é verificada pelo combinador de aspectos, no momento de compilação das customizações.

O procedimento descrito acima se aplica quando o ponto de extensão é definindo no projeto do produto. Outra situação possível ocorre quando uma customização exige a criação de um novo ponto de extensão que não faz parte do planejamento do produto. Isto pode ocorrer, por exemplo, quando um pedido de customização exige a criação imediata de uma nova JPI, sem possibilidade de esperar uma nova versão do *software*.

O procedimento para a criação de pontos de extensão não planejados é mostrado nas figuras 9 e 10. O responsável pela criação desses pontos de extensão não é o cliente, e sim uma equipe de suporte do fabricante. Primeiro se cria uma nova JPI para expor o ponto de extensão. Como não é possível nem permitido que o cliente altere o produto, é criada uma classe adaptadora na camada intermediária. Ela faz a ligação entre JPI e classes do produto. A figura 9 ilustra o caso mais simples, no qual a JPI serve para expor pontos de junção do produto. A classe adaptadora conteria somente a definição do ponto de corte exposto.

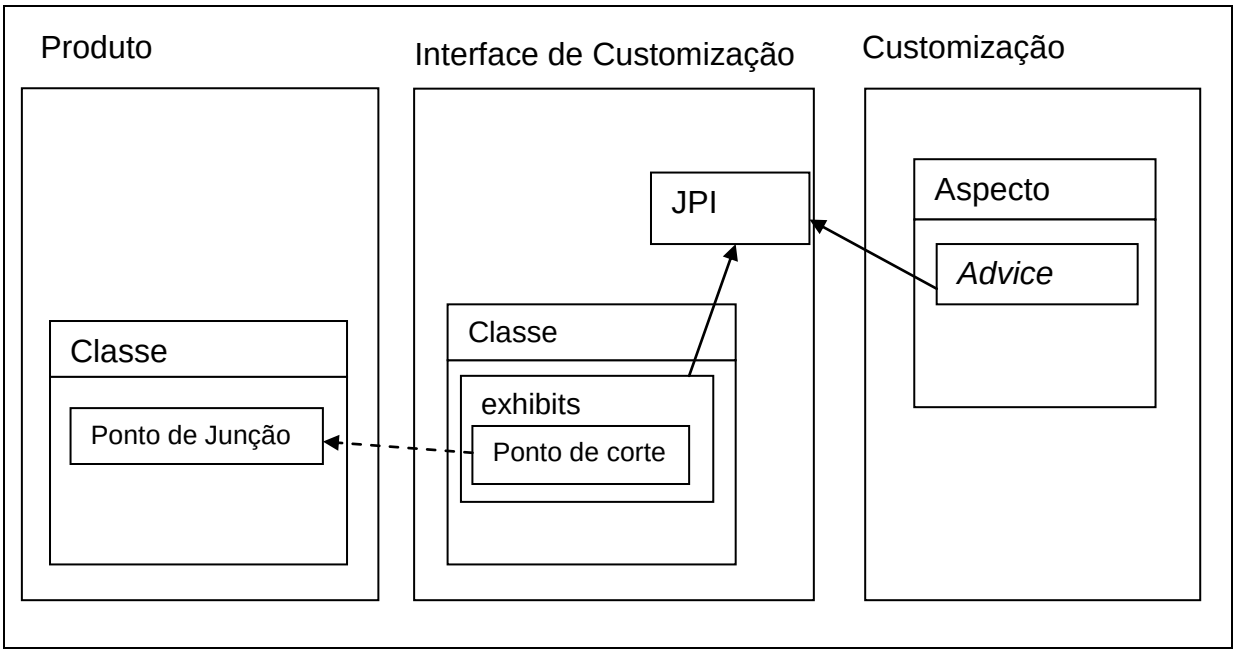


Figura 9 - Criação de ponto de extensão simples

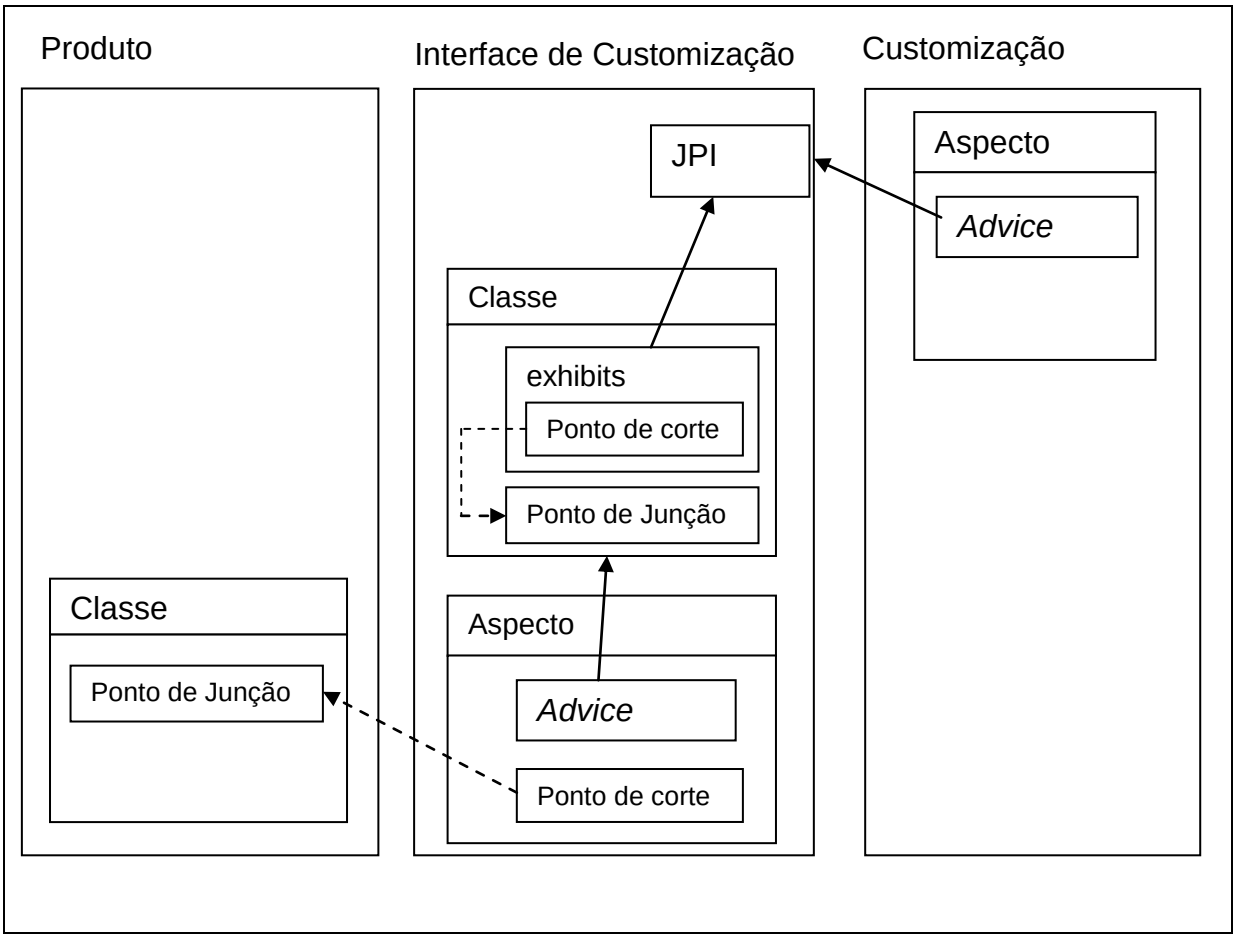


Figura 10 - Criação de ponto de extensão complexo

A figura 10 mostra uma situação mais complexa, quando a JPI não pode ser mapeada diretamente em um ponto de corte. Nesse caso, a classe adaptadora que expõe uma interface simplificada para a JPI e encapsula uma lógica mais elaborada para encaixar a customização ao produto. Um aspecto é criado para injetar o código da classe adaptadora nas classes do produto.

O uso de classes adaptadoras traz de volta o problema da fragilidade dos pontos de corte. As setas tracejadas na figura 10 mostram a dependência implícita/frágil entre “interface de customização”e “produto”. É o preço a se pagar pela flexibilidade de injetar customizações em qualquer parte do software.

Em resumo, camada de “Interface de customização” foi projetada para atender a requisitos conflitantes. Por um lado, há a necessidade de um contrato bem definido entre customizações e produto, para que ambos possam evoluir independentemente. Isto é conseguido pelo uso de JPIs na comunicação entre “interface de customização” e “customização”. Por outro lado, é preciso permitir a criação de novos pontos de extensão de forma rápida, sem que o produto seja alterado. Isto é obtido com o uso de classes adaptadoras, que não usam JPIs para se comunicar com o produto e que introduzem dependências frágeis. É responsabilidade do fabricante do produto gerenciar esses pontos de corte, os quais não são visíveis pelo cliente.

3.4. Gerenciamento da Interface de Customização

Há três equipes de desenvolvimento envolvidas no processo de customização:

a) Equipe do Produto

É a equipe de desenvolvimento do produto. Possui conhecimento profundo sobre a arquitetura do *software* e é a responsável por garantir a sua evolução. Seus desenvolvedores interagem com o *ACF* de duas formas:

- Criando novas JPIs.

- Atualizando os pontos de corte quando a evolução do produto exigir.

O foco da equipe de produto é programar pensando nas funcionalidades gerais do produto, que se apliquem a todos os usuários/clientes. Não é sua função atender a necessidades de um cliente específico.

b) Implantadores

É um programador responsável por implantar o produto em clientes. Não faz parte da equipe de desenvolvimento nem tem relações com ela. Possui pouco conhecimento sobre a arquitetura interna do produto. O implantador não pode criar JPIs novas, mas deve ter autonomia para criar customizações sem apoio da equipe de desenvolvimento do produto. Os ajustes que ele fizer devem continuar funcionando mesmo que o produto evolua. Sua interação com o *ACF* se resume a:

- Criar código de customização (aspectos).

c) Suporte

É a equipe do fabricante do produto responsável por ajustar a interface de customização. O suporte tem permissão para criar JPIs novas, classes adaptadoras e aspectos, e incorporá-las à camada de interface de customização. Ele também tem conhecimento profundo sobre a arquitetura do produto, e mantém contato com a equipe de desenvolvimento do produto. Ele interage com o *ACF* de três formas:

- Criando novas JPIs.
- Criando classes adaptadoras.
- Criando código de customização (aspectos).

Quando um cliente pede uma alteração de funcionalidade que não pode ser atendida por customizações nas JPIs existentes, o implantador responsável aciona a

equipe de suporte solicitando a criação de uma nova JPI. O suporte pode criar um *patch* (ajuste ou correção) contendo as classes adaptadoras e os aspectos necessários para a nova JPI. Desta forma, o cliente recebe as alterações pedidas de forma rápida e o código do produto não sofre alterações. Todo o código escrito pelo suporte se restringe às camadas de “interface de customização”. Em uma etapa posterior, a equipe de produto pode mover o código criado pelo parceiro para dentro do produto, de uma forma mais organizada.

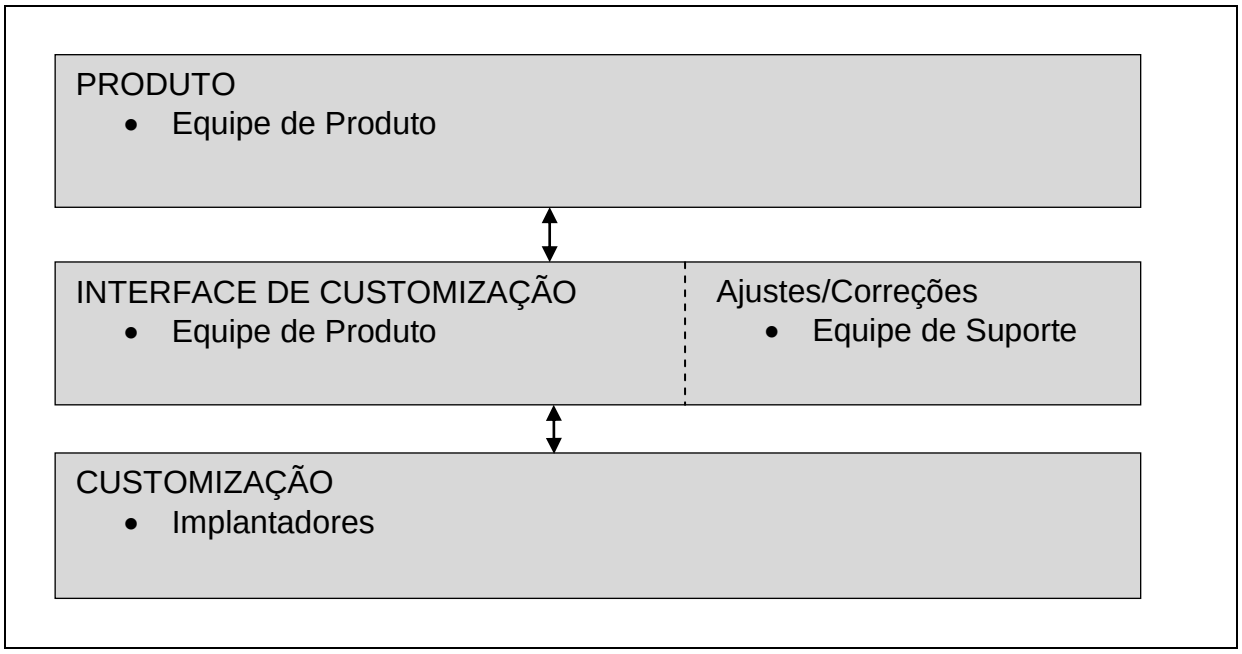


Figura 11 - Equipes responsáveis por cada camada

Os *patches* pedidos por um cliente são distribuídos somente para ele. Não são enviados para outros clientes. A equipe de suporte mantém um banco de dados com todos os *patches* criados. Para cada nova versão do produto há também novas versões dos *patches*. Eles são revisados e atualizados. Há três formas de tratamento das JPIs dos *patches* em uma nova versão:

a) São movidas para o produto. Nesse caso, a JPI do *patch* passa a fazer parte da interface de customização comum a todos os clientes. As classes adaptadoras são eliminadas ou incorporadas ao produto.

b) São mantidas na nova versão do *patch*. A JPI é muito específica ou a equipe de produto julga que não vale a pena movê-la para o produto nesse dado momento.

c) São depreciadas (*deprecated*) e futuramente removidas. Se os arquitetos do produto avaliam que a JPI em questão não se adéqua ao projeto do produto, eles podem criar uma JPI alternativa, mais aderente à arquitetura do produto e recomendar aos clientes que passem a usá-la num futuro próximo.

O objetivo em longo prazo é zerar quantidade de JPIs nos *patches*, para que eles deixem de ser necessários. *Patches* são criados para serem somente ajustes temporários.

3.5. Construção do Protótipo

O protótipo é composto duas bibliotecas Java. A biblioteca **br.ipt.jpi** define as anotações @JPI, e a biblioteca **br.ipt.acf** contém as classes do *framework* ACF.

3.5.1. Biblioteca de Anotações @JPI

A biblioteca **br.ipt.jpi** contém a definição das anotações `@Jpi` e `@Exhibits`, que são usadas para definir as JPIs. Também contém um validador de anotações, baseado no padrão JSR 269 (*Pluggable Annotation Processing API*). O validador é chamado em tempo de compilação para verificar se os pontos de corte expostos pelas anotações `@Exhibits` são compatíveis com as JPIs às quais eles estão associados.

3.5.2. Biblioteca do ACF

A biblioteca **br.ipt.acf** contém as classes do *framework* ACF. O protótipo usa como base a versão 3.0 do *framework* Spring, de código aberto (www.springsource.org). O Spring foi criado inicialmente para ser uma alternativa

aos *containers* J2EE. Seu propósito é facilitar a criação de aplicações corporativas, oferecendo as seguintes funcionalidades:

Container de componentes	Gerencia o ciclo de vida dos objetos Java com um mecanismo baseado em inversão de controle
Acesso a dados	Oferece ferramentas para trabalhar com banco de dados usando JDBC ou mapeamento objeto-relacional.
Gerenciamento de transações	Automatiza o gerenciamento de transações com bancos de dados.
Programação Orientada a Aspectos	Provê infraestrutura para a codificação de aspectos.
Aplicações *Web*	Oferece um framework MVC para a criação de páginas *web* e *web services*.
Outras	Segurança, testes, gerenciamento remoto, etc.

As razões para a escolha do Spring como base para construção do protótipo foram as seguintes:

- O Spring é um *framework* popular, o que torna o protótipo mais próximo da realidade de empresas de *software*. As funcionalidades oferecidas pelo Spring são de uso comum em aplicações corporativas (por exemplo, acesso a banco de dados e gerenciamento de transações). Isto facilita a etapa do experimento, na qual trechos de código de aplicações reais criadas com Spring serão copiados e adaptados em uma aplicação de teste do *framework* de customização.
- O Spring traz suporte nativo à programação orientada a aspectos, e aceita a notação @AspectJ. O Spring oferece duas implementações alternativas de POA. Uma é baseada no compilador e combinador de aspectos AspectJ Eclipse. A outra, chamada **Spring AOP**, é baseada em *proxies*. Nesta segunda opção, ao invés de injetar o código nas classes, o Spring cria um *proxy* para cada objeto. Ao invés de instanciar os objetos originais, o container do Spring instancia os *proxies*, que têm a função de chamar os *advices* antes e depois de repassar as chamadas aos métodos dos objetos originais.
- O Spring possui uma arquitetura aberta e pode ser facilmente configurado ou estendido. Como seu código é aberto, pode ser alterado para aceitar formas alternativas de processamento dos aspectos.

A biblioteca **br.ipt.acf** estende o Spring para que ele interprete as anotações `@Jpi` e `@Exhibits` e crie os *advices* nos pontos apropriados.A estratégia escolhida foi substituir o gerador de *proxies* do Spring AOP. A classe principal do Spring APO é `AspectJAwareAdvisorAutoProxyCreator`, que é responsável por criar os *proxies* dos objetos Java, verificar se há aspectos a serem adicionados e configurar os *proxies* para que façam a chamada aos *advices* apropriados. Esta classe foi substituída por uma nova `JpiAwareAdvisorAutoProxyCreator`, que leva em conta também as anotações de JPI ao adicionar as chamadas aos *advices*.

4 TESTE DO PROTÓTIPO

Para testar a aplicação do ACF, foi feito um levantamento em uma empresa fabricante de *softwares* corporativos, em busca dos tipos mais frequentes de customizações que levam a alterações de código nos produtos.

A partir desse levantamento, foi criada uma coleção de fragmentos de código extraídos de produtos reais, com o propósito de reproduzir a estrutura dos casos típicos de customização. O ACF foi aplicado a essa coleção, que em seguida sofreu alterações simulando de uma evolução do software.

Esta seção é dividida em quatro partes. O item 4.1 apresenta uma visão geral do processo adotado pela empresa para atender a pedidos de customização de produtos. O item 4.2 descreve a forma como os produtos expõem pontos de extensão. O item 4.3 mostra um levantamento quantitativo dos tipos de customização mais pedidos pelos clientes. O item 4.4 descreve o procedimento de teste do protótipo, feito com base nos casos de customização encontrados no levantamento.

4.1. Processo de customização

Foram feitas entrevistas com os gerentes responsáveis pelas equipes de três produtos. O objetivo foi identificar como as equipes recebem os pedidos de suporte relativos a customizações e como são feitas as alterações no produto para tornar as customizações possíveis.

Os produtos pesquisados seguem uma arquitetura de três camadas:

- Banco de dados
- Aplicação
- Apresentação

Os bancos de dados são do tipo relacional, baseados na linguagem SQL (*Structured Query Language*). As tabelas, gatilhos, procedimentos armazenados e funções são criados pelos *scripts* de instalação do produto. A camada de aplicação

normalmente é instalada em um servidor de aplicação Java, que provê serviços web (*Web Services*) ou páginas HTML com suporte a AJAX. A camada de apresentação é composta de navegadores web, que mostram as páginas HTML, clientes em PC e aplicativos móveis para telefone celular ou *tablet*, que acessam os serviços web expostos pela camada de apresentação.

Os tipos mais comuns de pedidos de customização feitos pelos clientes são:

- Criação de telas novas na interface gráfica, com os propósitos de:
    - Adicionar funcionalidades que o produto não possui, mas que são necessárias para o cliente.
    - Apresentar uma interface gráfica mais adaptada ao modo de operação do cliente, sem a criação de novas funcionalidades.
- Alteração de telas existentes
    - Adição de campos novos.
    - Ajustes de layout (posicionamento de campos, títulos de campos, ocultamento de campos ou abas).
- Criação de novos relatórios
    - Para personalizar a apresentação dos dados.
    - Para incorporar dados de outras fontes além do próprio produto.
- Ajustes em regras de negócio
    - Validações em geral. Em particular validações adicionais em campos de telas e em registros de banco de dados.
    - Alterações em algoritmos de processamentos de dados em tela ou em lote, para adequar ao produto a particularidades do cliente.
    - Adição de automatismos, ou seja, criação de novas rotinas ou processos extras, a serem usados em telas ou relatórios novos.

Os ajustes podem estar associados a alterações na estrutura do banco de dados. Quando uma tela recebe campos novos, isso requer a adição de colunas em tabelas do banco de dados, alteração em relatórios e em regras de validação.

Além disso, é comum os ajustes serem inter-relacionados. Por exemplo, quando se adiciona um novo campo em tela, podem ser necessários ajustes em regras de negócio para processar esta informação adicional, e pode ser preciso também criar novos relatórios que incluam o campo extra.

Os pedidos de customização são avaliados pelas equipes de produtos, que elaboram as propostas de serviços para aprovação do cliente. Essas propostas detalham a forma como serão feitos os ajustes. Há basicamente três formas principais de ajuste: parametrização/configuração, extensão do produto ou alteração do produto.

a) Parametrização ou configuração.

Os produtos são bastante configuráveis. Como a base de clientes é ampla, o *software* é projetado de antemão para que os ajustes mais comuns possam ser feitos com alterações simples em tabelas de configuração. O *layout* das telas, por exemplo, pode ser modificado por parametrização, permitindo alteração de posição de campos, textos, e validações simples. As regras de negócio podem ser alteradas com *scripts* ou fórmulas cadastradas em tabelas de configuração.

Quando o ajuste é feito por parametrização, o fabricante do produto garante a sua compatibilidade com versões futuras do *software*.

b) Extensão do Produto

Quando a parametrização não é suficiente, é possível estender o produto, adicionando novas funcionalidades. Os produtos podem ser estendidos de duas formas:

Novos artefatos: é possível criar telas, rotinas e relatórios adicionais, feitos para atender um cliente específico. Esses artefatos funcionam como adendos ao produto, mas não fazem parte dele. Também é possível desabilitar algumas telas ou relatórios do *software* original e substituí-los pelos artefatos novos. Os módulos adicionais são de propriedade do cliente, e é sua responsabilidade mantê-los atualizados. O fabricante só garante manutenção desses artefatos quando há um contrato prevendo esse serviço.

Pontos de extensão: o fabricante oferece pontos de extensão na forma de *hooks*, que expõem parte da API do produto. Os clientes podem criar rotinas e registrá-las em pontos de extensão específicos, para mudar o funcionamento do produto. Há pontos de extensão para validações de inserção, alteração ou remoção de registros no banco de dados. Também há pontos de extensão para modificar consultas a banco de dados e funções de cálculo. E finalmente, há pontos de extensão para alterar o funcionamento de rotinas complexas. O fabricante documenta o contrato dos pontos de extensão, definindo as pré-condições, pós-condições e os parâmetros de entrada e saída, e garante que esse contrato será mantido em versões futuras. Mas é de responsabilidade do cliente a manutenção das rotinas de extensão.

c) Alteração do Produto.

Quando o pedido do cliente não pode ser atendido por mudanças na parametrização nem pelos mecanismos disponíveis de extensão do produto, é necessário alterar o *software*, adicionando funcionalidades novas que ficarão disponíveis para toda a base de clientes em versões futuras. A decisão sobre quando um pedido de alteração deve ser incorporado ao produto fica a cargos dos gerentes de produto e dos membros mais experientes da equipe. Alguns dos critérios usados na tomada de decisão são:

- Se não for possível atender o cliente com mudanças na parametrização nem pelos mecanismos disponíveis de extensão do produto.
- Se vários clientes estão pedindo a mesma modificação, ou se a equipe prevê que outros clientes pedirão num futuro próximo. Nesse caso, pode ser mais incorporar essa modificação ao produto ao invés de criar customizações semelhantes para vários clientes.
- Quando o pedido do cliente faz com que a equipe identifique uma nova possibilidade de melhoria do produto que lhe trará uma vantagem competitiva.

É importante que as alterações do produto não causem uma quebra de compatibilidade com versões anteriores. As estratégias para evitar este problema são:

- Adicionar módulos ou funcionalidades totalmente novas, que não afetem os módulos antigos mesmo que dependam deles.
- Alterar módulos ou rotinas antigas. Se a mudança tiver o potencial de causar quebra de compatibilidade, adiciona-se um parâmetro de configuração no sistema para habilitar ou desabilitar a nova funcionalidade. Por padrão este parâmetro estará desabilitado, fazendo com que o sistema funcione da forma antiga até que o cliente decida chavear a configuração.
- Adicionar pontos de extensão. Neste caso, a mudança no produto é relativamente pequena, sendo feita somente a injeção de *hooks* em pontos estratégicos do software. Toda a lógica de customização fica fora do produto, em rotinas a serem mantidas pelos implantadores ou pelo cliente.

De forma geral, quando os parâmetros ou pontos de extensão existentes não permitem atender ao pedido do cliente, criam-se novos parâmetros ou pontos de extensão, que são liberados em *patches* (correções rápidas) e/ou versões futuras. Nas primeiras versões do produto, dava-se preferência à criação de novos parâmetros. Com o tempo e com o *feedback* dos clientes e implantadores, notou-se que a criação de muitos parâmetros tornava a manutenção do produto mais custosa e a parametrização mais complicada. Além disso, o uso de parâmetros não dá aos clientes a mesma flexibilidade e o controle dos detalhes dos ajustes que os pontos de extensão proporcionam. Com isso, nas versões mais recentes dos produtos, tem se dado preferência à criação de pontos de extensão como mecanismo de customização.

4.2. Tipos de Pontos de Extensão

Foi feita uma inspeção no código dos produtos para se determinar quais tipos de pontos de extensão são oferecidos e de que forma eles são implementados.

4.2.1. Forma de Implementação - *Hooks*

Os produtos analisados expõem pontos de extensão na forma de *hooks*. *Hooks* são basicamente pontos no código do produto onde há chamadas para métodos escritos por terceiros.

Em Java, *hooks* podem ser construídos com o uso do padrão de projeto *Template Method*. A listagem 16 mostra um exemplo simplificado de implementação. A classe `ProdutoComHook`, que faz parte do produto, contém o método `rotina()`, que executa uma regra de negócio padrão do sistema. O método `rotina()` está preparado para receber customizações. Logo no início do seu código, na linha 11, é feita uma chamada ao método `preRotina()`, que é o *hook*. Este é um método customizado, e sua implementação é feita na classe `CustomizacaoRotina` (listagem 18). Seu código não faz parte do produto e é escrito para um cliente específico. Quando `rotina()` é chamado, ele executa primeiro o código customizado `preRotina()`, e só depois executa o resto do seu código.

```
01 public class ProdutoComHook
02 {
03     private HookPreRotina hookPreRotina;
04
05     /**
06      * Rotina com hook
07      */
08     public void rotina()
09     {
10           //Hook executado antes do corpo da rotina,chamando regras customizadas
11           hookPreRotina.preRotina();
12
13           //Corpo da rotina. Aqui é executada a lógica padrão do produto
14           .....
15           .....
16
17     }
18
19     /**
20      * Método para registrar o hook
21      * @param hookPreRotina Hook executado antes do corpo da rotina
22      */
23     public void registrarHook(HookPreRotina hookPreRotina)
24     {
25           this.hookPreRotina=hookPreRotina;
26     }
27
28 }
```

Hook

Listagem 16 - Exemplo de chamada a hook

```
public interface HookPreRotina {
	public void preRotina();
}
```

Listagem 17 - Exemplo de interface *template* de *hook*

```
public class CustomizacaoRotina implements HookPreRotina {

	public void preRotina() {
		System.out.println("Hook executado");
	}

}
```

Listagem 18 - Exemplo de classe concreta de *hook*

O método `registrarHook()` é o responsável for fazer a associação do código customizado com o *hook*. Imediatamente após a criação de uma instância de `ProdutoComHook` é preciso que o *software* crie uma instância de `CustomizacaoRotina` e passe ela como parâmetro para `registrarHook()`.

A interface `HookPreRotina` (listagem 17) é o *template* que define a assinatura dos métodos *hook*.

4.2.2. Classificação dos Pontos de Extensão

Para facilitar o entendimento de como os pontos de extensão são usados nos produtos, eles foram classificados seguindo dois critérios distintos: estrutural e funcional.

a) Estrutural

O critério estrutural serve para classificar os *hooks* de acordo com a posição onde eles são injetados dentro do código e de como o seu valor retorno é usado pela rotina principal. Do ponto de vista estrutural, há três tipos de *hooks*: de substituição, de alteração e interno. A tabela 4 dá uma breve descrição desses tipos.

Tabela 4 - Classificação Estrutural dos Pontos de Extensão

Tipo	**Descrição**
Substituição	É usado quando se deseja que um método customizado substitua um método padrão do produto. Nesse caso, só o método customizado é executado e o seu valor de retorno é usado no lugar do retorno do método padrão. O método padrão NÃO é executado.
Alteração	Usado quando se deseja que um método customizado altere o resultado de um método padrão do produto. O método padrão é executado primeiro. Ao final de sua execução é feita uma chamada ao *hook*, que pode ou não alterar o valor de retorno do método principal.
Interno	É um *hook* injetado no meio do código de um método. É usado para permitir a customização de passos intermediários de rotinas.

Exemplos de código são apresentados no apêndice A.

b) Funcional

O critério funcional classifica os pontos de extensão de acordo segundo o tipo de operação que eles afetam. Após a inspeção do código fonte dos produtos, foram identificadas as situações mais comuns.

Tabela 5 - Classificação Funcional dos Pontos de Extensão

Tipo	**Descrição**
Consulta	Pontos de extensão que alteram o resultado de consultas a banco de dados.
Edição de Registro	Pontos de extensão que afetam operações de inserção, alteração ou remoção de registros no banco de dados.
Função	Pontos de extensão que alteram o resultado de rotinas de cálculo.
Outros	Pontos de extensão que não se enquadram nas categorias anteriores.

Consulta

Os três produtos analisados fazem uso intensivo de bancos de dados relacionais. As consultas a banco de dados, feitas normalmente com comandos "SELECT" em linguagem SQL, são frequentes. Os registros obtidos em consultas são usados para preencher campos de telas, montar relatórios, montar páginas HTML e retornar dados de *web services*. Algumas customizações exigem a alteração das consultas para adicionar campos extras ou filtrar seus registros com base em informações de sistemas legados ou externos. Consequentemente, vários métodos de consulta oferecem pontos de extensão para alterar os dados retornados.

Edição de registro

Outra operação comum é a alteração de registros em banco de dados. Os métodos que efetuam operações de inserção, alteração ou remoção de registros no banco de dados oferecem dois *hooks*. Um é executado executados antes da operação, e o outro depois. Esses pontos de extensão normalmente são usados para mudar valores de campos, modificar as regras de validação dos registros ou efetuar outras alterações no banco de dados.

Função

Funções aqui são entendidas como rotinas de cálculo, que recebem um ou mais parâmetros e retornam um valor numérico ou textual. Podem ser métodos escritos em Java ou funções definidas em SQL. Alguns exemplos de funções: cálculo de média de notas acadêmicas, cálculo dos dígitos de um boleto bancário, identificação do setor ao qual um funcionário pertence, cálculo de desconto de uma dívida para pagamento imediato. É comum que pedidos de customização exijam alterações nos cálculos executados por funções. A alteração pode ser a substituição completa da função por outra feita especialmente para um cliente, ou com a alteração parcial do seu algoritmo.

Outros

São pontos de extensão que não se enquadram nas categorias anteriores. Na maioria dos casos servem para modificar rotinas complexas, que afetam registros de várias tabelas do banco de dados simultaneamente. Essas rotinas normalmente são constituídas de uma sequência de passos, havendo pontos de extensão distribuídos nessa sequência. Exemplo: cancelamento da matrícula de um aluno em uma instituição acadêmica. O cancelamento envolve alterações nas vagas das turmas nas quais o aluno estava matriculado, alterações no histórico escolar, alterações nas cobranças financeiras e nova geração de boletos. Pedidos de customização podem levar à inclusão de novos passos ou alteração em resultados parciais de processamento.

Existe uma correlação entre as classificações funcional e estrutural. Para cada tipo de operação, há uma forma preferida de criação de pontos de extensão, mostrada na Tabela 6.

Tabela 6 - Relação entre tipos estruturais e funcionais

Tipo Funcional	**Tipos Estruturais**
Consulta	Substituição, alteração
Edição de Registro	Interno
Função	Substituição, alteração
Outros	Interno (maioria), substituição, alteração

Para consultas e funções, são oferecidos pontos de extensão de substituição e alteração. Assim, os clientes tem a opção de fazer pequenos ajustes nos resultados de consultas ou cálculos, ou substituir totalmente a lógica padrão por um algoritmo próprio.

Métodos de edição de registro oferecem dois pontos de extensão internos. Um é executado executados antes da operação, e o outro depois.

Nos outros casos, prevalece o uso de *hooks* internos, mas também há casos de pontos de extensão de substituição e alteração.

Exemplos de código são dados no apêndice B.

4.3. Levantamento quantitativo de customizações

Com o objetivo de identificar os tipos de ajuste mais frequentes, foi coletada uma amostra de 90 documentos com propostas de customização, feitas para três produtos nos últimos oito anos. Os três produtos, identificados como A, B e C, diferem no grau de adoção do conceito de ponto de extensão. O produto A foi o primeiro a incluir pontos de extensão em sua arquitetura, portanto os possui em maior quantidade. No produto B a adoção foi relativamente recente e o número de pontos é menor. O produto C não oferece pontos de extensão.

Cada proposta detalha a necessidade de um cliente e os ajustes que serão necessários para atendê-la. A amostra é bastante heterogênea. Algumas propostas contém somente um ajuste no produto, outras incluem dezenas. No total, obteve-se uma lista de 502 ajustes, que foram classificados em quatro categorias:

- Novos Artefatos
- Novos campos em telas
- Customização de Pontos de Extensão
- Alterações no Produto

a) Novos Artefatos

A maioria dos pedidos de customização envolve a criação de componentes adicionais, feitos especialmente para o cliente. Os mais comuns são: tabelas de banco de dados, telas, relatórios, processos em lote e utilitários de integração. No total, são 338 itens, o que equivale a 67,3% do total de ajustes propostos.

b) Novo Campo em tela ou registro

A maior parte das telas dos produtos consiste em formulários para cadastro ou alteração de registros em banco de dados. É comum que clientes peçam a inclusão de campos novos nos formulários, para permitir o cadastro de informações adicionais. Por exemplo, se um cliente deseja controlar quais dependentes dos seus funcionários recebem auxílio de creche, e o produto não trata essa informação, será necessário alterar a tela de cadastro de dependentes dos funcionários, adicionando um campo do tipo *checkbox* chamado “Auxílio Creche”.

A criação de um campo adicional normalmente exige alterações nas três camadas do produto: banco de dados, aplicação e interface gráfica. Há modificações no layout da tela, adição de coluna em tabela do banco de dados e alteração das regras de negócio.

A criação de novos campos equivale a 10,6% dos ajustes pedidos.

c) Customização de ponto de extensão

Os pontos de extensão respondem por 12,5% dos ajustes. São usados quando o cliente quer mudar alguma regra de negócio do produto, ou acrescentar novas regras.

d) Modificações no Produto

As alterações no produto respondem por 9,6% dos itens de ajuste. Nesses casos, a equipe decidiu incluir no próprio produto funcionalidade pedida pelo cliente. Dessa forma, o pedido do cliente não é atendido por uma customização, mas sim por uma evolução do produto. As mudanças são liberadas em *patches* ou em versões regulares, para todos os clientes.

A tabela 7 mostra a quantidade de ajustes por produto, e as porcentagens em relação ao total. Os números entre parênteses indicam as ocasiões em que o produto não oferecia nenhum ponto de extensão apropriado para atender ao pedido de customização. Nesses casos foi preciso alterar o produto para definir um novo ponto de extensão, que seria posteriormente customizado para o cliente.

Tabela 7 - Total de ajustes por tipo e subtipo.

Tipo de ajuste		**Produto A**	**Produto B**	**Produto C**	**Total %**	**Total %**
Novo Artefato	Tabela	32	19	10	**12,2%**	**67,3%**
	Tela	39	70	20	**25,7%**	
	Relatório	17	82	6	**20,9%**	
	Processo em Lote	0	3	0	**0,6%**	
	Integração	1	30	2	**6,6%**	
	Outros	7	0	0	**1,4%**	
Novo campo em tela/registro		32	21	0	**10,6%**	**10,6%**
Customizar Ponto de extensão	Registro	29 (6)	2 (2)	0	**6,2%**	**12,5%**
	Função	6 (4)	2 (1)	0	**1,6%**	
	Consulta	3 (3)	1 (1)	0	**0,8%**	
	Geral	12 (0)	8 (8)	0	**4,0%**	
Modificações no Produto		0	15	33	**9,6%**	**9,6%**
Total		**178**	**253**	**71**	**100%**	**100%**

Aproximadamente dois terços dos ajustes (67,3%) são para a criação de novos artefatos. Isto indica que na maior parte dos casos os clientes querem incorporar novas funcionalidades no sistema, ao invés de alterar as que já existem.

O produto C foi o que recebeu maior quantidade de modificações de produto, justamente por ser o único dos três que não aceita pontos de extensão. O produto A, que está mais maduro na adoção de pontos de extensão, não precisou ser alterado.

4.4. Teste do Protótipo ACF

O propósito do *framework* ACF é facilitar customizações que envolvam a criação de novos pontos de extensão, modificando o comportamento do produto. Ele não se aplica à criação de novos artefatos, nem aos casos em que o produto sofre uma evolução (itens “a” e “d” da seção 4.3). Mas o ACF pode ser aplicado na customização de pontos de extensão e na criação de novos campos em telas (itens “b” e “c”).

O teste do protótipo tem dois objetivos. O primeiro é verificar se o ACF pode ser aplicado com sucesso em situações de personalização encontradas na prática. O segundo é comparar a abordagem baseada em interfaces orientadas a aspectos usada no com a técnica de *hooks* adotada nos produtos pesquisados.

Não foi possível para este trabalho usar trechos reais de código dos produtos analisados, por razões de propriedade intelectual. Foram criados exemplos adaptados do código real, que reproduzem as características mais relevantes do ponto de vista de customização.

O teste do protótipo foi executado em quatro etapas:

- Criação de uma aplicação de teste
- Inserção das customizações
- Evolução do software
- Ajuste das customizações

4.4.1. Criação da Aplicação de Teste

Foi construída uma aplicação que provê serviços web ao estilo REST (*Representational State Transfer*). A aplicação oferece ao todo 11 serviços web, que

executam operações de consulta, edição de registro, cálculo de funções e outras (tabela 8).

Tabela 8 - Quantidade de serviços por tipo de operação

Operação	**Qtde de Serviços**
Consulta	3
Função	2
Edição de Registro	2
Rotina Genérica	4

A estrutura da aplicação é mostrada de forma simplificada na figura abaixo. Como base para a construção da aplicação foram usados os *frameworks* Spring e Apache CXF. O Spring gerencia a injeção de aspectos, as transações do banco de dados e instanciação das classes. O Apache CXF provê a infraestrutura de serviços REST.

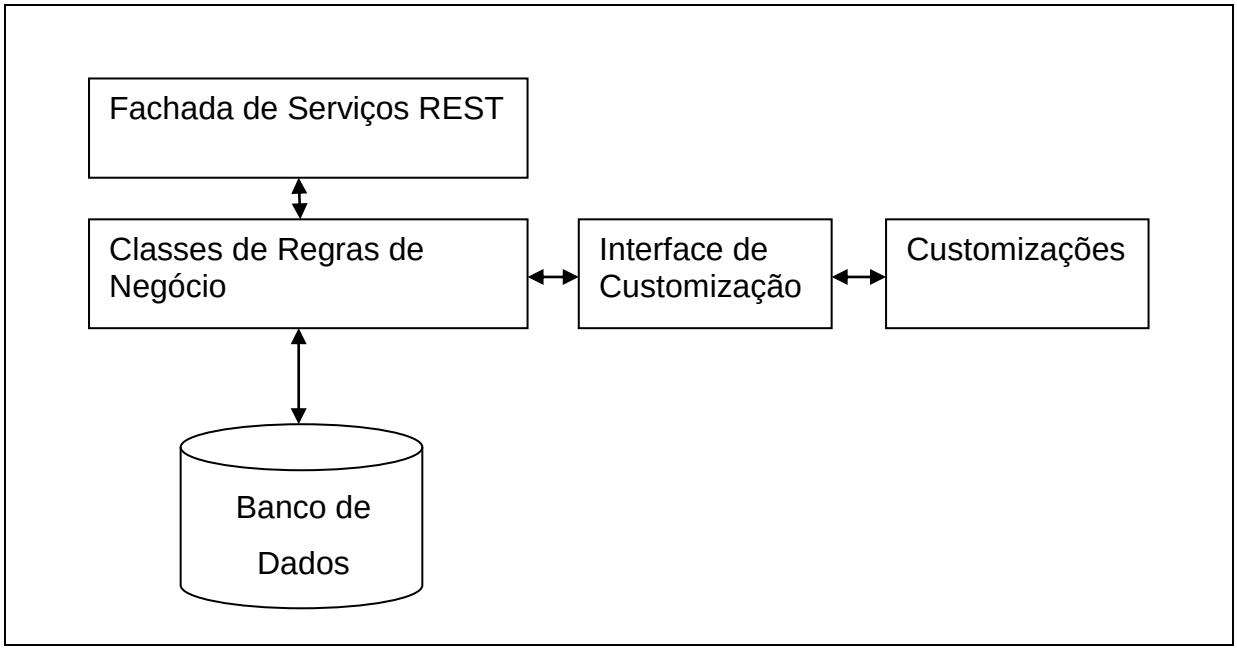


Figura 12 - Aplicação de Teste

A aplicação executa algumas operações de um sistema de gestão acadêmica. São consultas e alterações na situação de um aluno em uma instituição de ensino. O

banco de dados consiste um subconjunto simplificado das tabelas de um sistema de gestão acadêmica real. As regras de negócio também são versões simplificadas, mas que mantêm a estrutura básica da implementação real.

Foram construídas duas versões da aplicação. Uma delas foi preparada para receber customizações por *hooks*, e a outra pelo ACF. Ambas as versões contêm as mesmas classes, com ligeiras diferenças no código associadas à infraestrutura de customização. Na aplicação customizada por *hooks*, foi necessário criar interfaces *template* (como na listagem 17) e incluir as chamadas aos pontos de extensão no métodos (como na listagem 16). Na aplicação customizada com o ACF, foram incluídas as definições das JPIs e as anotações `exhibits` nas classes customizadas (de forma similar à listagem 15).

Exemplos de código são dados no apêndice B.

4.4.2. Inserção das customizações

Foram inseridas customizações típicas em cada uma das classes. Elas foram implementadas de duas formas, com *hooks* e com o ACF.

Tabela 9 - Customizações aplicadas ao corpo de teste

Consulta	Substituição da consulta original por outra. A consulta original não é executada.
	Alteração dos resultados da consulta original. A consulta original é executada, e os registros obtidos são alterados por um código de ajuste
Função	Substituição da função original por outra. A função original não é executada.
	Alteração do resultado da função. A função original é executada, e o resultado obtido é alterado por um código de customização
Edição de Registro	Adição de um campo extra no registro
	Alteração nos critérios de validação dos registros
	Execução de passos após uma operação de cadastro
Rotina Genérica	Inclusão de um passo extra, específico para o cliente
	Definir uma condição para que um passo seja executado.

As customizações listadas na tabela 9 são semelhantes àquelas encontradas nas 90 propostas analisadas no capítulo 4.

4.4.3. Evolução do produto

Uma nova versão da aplicação teste foi criada, com 9 alterações nas classes da aplicação. O propósito é simular a evolução do produto.

Tabela 10 - Alterações por evolução de produto

Consulta	Adição de novos parâmetros de entrada
	Alteração no algoritmo, mudando os critérios de busca
	Adição de campos extras no retorno
Função	Adição de novos parâmetros de entrada
	Alteração no algoritmo
Edição de Registro	Adição de campos novo nos registros
	Alteração nos critérios de validação dos registros
Rotina Genérica	Adição de novo parâmetros de entrada
	Alteração no algoritmo.

4.4.4. Ajustes das customizações

Depois da alteração do produto, foram feitos testes para verificar se as customizações continuaram funcionando. Verificou-se se elas continuavam sendo compiladas, se continuavam sendo executadas e se o resultado da execução continuava sendo o esperado.

5 RESULTADOS DOS TESTES

Ao longo dos testes, o código das duas aplicações-teste foi analisado para verificar se o ACF conseguia cobrir todos os tipos de customização encontrados nos produtos pesquisados. Também foi feita uma comparação com o propósito de identificar as vantagens e desvantagens do ACF em relação à técnica baseada em *hooks*.

5.1. Execução dos Testes

A seguir se descreve a forma como os passos dos testes foram executados, e o comportamento das customizações.

5.1.1. Inserção das Customizações

Foram criados pontos de extensão para reproduzir os mesmos tipos de customização encontrados na implementação com hooks. Os pontos de extensão foram criados da seguinte forma:

- Criação do Ponto de Extensão no Produto
    - Criou-se uma JPI para cada ponto de extensão, na forma de um aspecto com anotação `@Jpi`.
    - Adicionou-se uma anotação `@Exhibits` para cada classe que recebeu o ponto de extensão. As propriedades `jpi` e `pointcut` da anotação fazem a ligação entre a JPI e a classe estendida. A propriedade `jpi` define o nome da JPI associada ao ponto de extensão e a propriedade `pointcut` define os métodos a serem estendidos.
- Criação da Customização
    - Criou-se um aspecto contendo somente advices. Os pontos de corte dos *advices* apontam para a JPI.

O ACF foi capaz de cobrir todos os tipos de customização encontrados nos produtos. De modo geral, *advices* do tipo `@Around` foram suficientes para cobrir todos os tipos de ponto de extensão. A forma como as JPIs foram introduzidas no produto variou de acordo com o tipo de operação.

a) Consultas e Funções

Consultas e funções recebem *hooks* de substituição e alteração. A implementação equivalente com ACF é o uso de advices do tipo `@Around`. A ideia inicial, quando o código das customização começou a ser desenvolvido, era usar advices `@Around` para substituição e `@After` para alteração. Mas a linguagem AspectJ oferece a classe auxiliar `thisJoinpoint`, que permite que o método da customização faça uma chamada ao método original que está sendo estendido. Os *hooks* (pelo menos nos produtos analisados) não oferecem esta opção. Com o uso desta classe auxiliar, *advices* `@Around` puderam ser usado para implementar simultaneamente ajustes de substituição e alteração.

b) Edição de Registro

Operações de edição de registro recebem *hooks* internos, executados imediatamente antes e depois da operação em si (ver apêndice B). A implementação equivalente com o ACF exigiu que fossem incluídas chamadas a métodos vazios dentro do método principal, para servirem de marcadores para a injeção dos aspectos. Advices do tipo `@Around` foram usados para substituir esses métodos vazios por métodos customizados.

Nesta situação, a estrutura do código do produto com ACF não é muito diferente da versão com *hooks*. Ambos precisam que o código do produto chame explicitamente os métodos de customização. O que muda é o mecanismo usado para injetar o código personalizado.

c) Rotinas genéricas

As rotinas genéricas também receberam *hooks* internos. Foram customizadas com *advices* `@Around` e métodos marcadores vazios, da mesma forma que a operação de edição de registros.

Todas as customizações foram criadas em um projeto Java independente do produto, compilado em um arquivo separado (um binário de extensão JAR).

5.1.2. Inserção de customização não prevista no produto

O ACF foi projetado para permitir a criação de novos pontos de extensão sem necessidade de alterar o produto. Isto é feito através de aspectos e classes adaptadoras.

Foram feitos dois testes. O primeiro consistiu na criação de um ponto de extensão para substituir uma função. Seria o equivalente a criar um *hook* de substituição seguindo a estrutura mostrada na figura 9. O outro teste foi feito em rotina genérica composta de uma sequência de passos, na qual foi incluído um passo extra customizado. Equivale a criar um *hook* interno, seguindo o esquema da figura 10.

Nas duas situações testadas, o *framework* funcionou adequadamente, e as customizações foram executadas com êxito.

5.1.3. Evolução da aplicação

Para simular uma evolução do produto, foi criada uma nova versão da aplicação teste, com as alterações listadas na tabela 10. Depois das alterações, foi preciso fazer alguns ajustes adicionais no produto para que as customizações continuassem funcionando. Os ajustes variaram de acordo com o tipo de alteração. A seguir são descritos os ajustes aplicados em cada caso.

a) Adição de novos parâmetros de entrada

Algumas classes da aplicação-teste tiveram métodos alterados para receber mais um parâmetro de entrada. Como consequência, a assinatura do método mudou, afetando hooks e JPIs. O quadro abaixo compara os ajustes que foram exigidos manter as customizações funcionando, com *hooks* e com o ACF.

Tabela 11 - Ajustes devidos à inclusão de novo parâmetro

Operação	Ajustes (hooks)	Ajustes (ACF)
Consulta	Nenhum	Alteração do ponto de corte da anotação `@Exhibits`
Função	Nenhum	Alteração do ponto de corte da anotação `@Exhibits`
Edição de Registro	Nenhum	Nenhum
Rotina genérica	Nenhum	Nenhum

A anotação `@Exhibits` define qual método será alterado ou substituído por um *hook* com o uso de uma expressão de ponto de corte. Se a assinatura desse método mudar, é preciso que o ponto de corte seja atualizado para refletir essa mudança. Caso contrário, o ACF não injetará o código da customização.

b) Alteração em algoritmo

Em todos os casos testados, mudanças no algoritmo do produto não afetaram estruturalmente as customizações, isto é, não impediram que o código customizado fosse executado. Isto é uma decorrência natural de se trabalhar com interfaces, sejam elas orientadas a objetos (nos hooks), ou orientadas a aspectos (no ACF). Como as interfaces se mantiveram inalteradas entre uma versão e outra, não houve quebra de compatibilidade e portanto não foram necessários ajustes.

c) Alterações em operação de edição de registro

Foram testados dois tipos de alteração nas operações de cadastro de registro. A primeira foi a inclusão de um campo adicional no registro. Isto ocorre com frequência, como consequência da adição de registros em telas. Cada vez que um campo é adicionado em uma tela, é necessário criar uma coluna extra na tabela correspondente no banco de dados, e ajustar as classes que fazem o cadastro dos

registros para tratar desse novo campo. Nos testes, essa alteração não afetou a execução do código customizado. O campo novo foi simplesmente ignorado.

A segunda alteração foi de mudança nas regras de validação de registros. Também nesse caso, alteração não afetou a execução do código customizado.

d) Adição de colunas no retorno de consultas

As colunas extras são ignoradas pelo código customizado, que continuou funcionando.

e) Alteração não prevista em rotinas genéricas

Foram alteradas as duas rotinas que receberam customizações não previstas no produto, citadas no item 5.1.2. Nos dois casos, foram adicionados parâmetros extras na assinatura dos métodos, o que invalidou os pontos de corte das anotações `@Exhibits` e impediu que o código das customizações fosse chamado.

A solução para esse problema foi mover o código das classes adaptadoras para dentro do produto, conforme previsto no item 3.4.

5.2. Comparação entre ACF e *hooks*

De forma geral, foi possível usar o AFC para criar pontos de extensão em todos os casos onde isso era feito com o uso de hooks. Isto significa que nas situações testadas neste trabalho, as duas técnicas são igualmente aplicáveis. Mas o uso de aspectos pelo ACF traz algumas vantagens e desvantagens em relação aos *hooks*, que serão discutidas a seguir.

5.2.1. Cobertura de Pontos de Extensão não previstos

O ACF foi projetado para permitir a criação de novos pontos de extensão, sem necessidade de alterar o produto original. Isto é feito com as classes adaptadoras. Logo, por projeto, ele é superior nesse quesito à abordagem baseada

em *hooks*. Usando *hooks*, novos pontos de extensão só podem ser criados em versões novas do produto.

Por outro lado, a análise das propostas de customização mostra que a necessidade de criação de pontos de extensão novos tende a diminuir com a maturidade do produto. A tabela 7 mostra entre parênteses o número de vezes em que foi preciso criar um novo ponto de extensão. No produto A, no qual o uso de hooks é adotado há mais tempo, de um total de 50 pontos de extensão customizados, somente 15 eram novos. No produto B, com menos tradição no uso de *hooks*, de 13 pontos de extensão, 12 eram novos.

Além disso, com um planejamento adequado, é possível liberar *patches* frequentes do produto com a inclusão de *hooks* novos de acordo com a demanda dos clientes.

5.2.2. Simplicidade do código

O ACF traz as vantagens de POA no que se refere à modularização das funcionalidades transversais. O uso do ACF simplificou o produto, dispensando as chamadas a *hooks* que poluíam código das rotinas. A definição do ponto de extensão exposto pelas classes foi movida para os pontos de corte descritos nas anotações `@Exhibits`. Descrevendo de outra forma, ACF trata a customização de código como uma funcionalidade transversal, e ao usar POA reduz o emaranhamento do código decorrente dela. Relembrando, emaranhamento é quando um único método possui código para tratar de dois ou mais objetivos distintos, o que torna o objeto menos coeso.

Uma desvantagem do ACF é que a forma de programar usando anotações @AspectJ pode ser confusa para desenvolvedores não familiarizados com POA. Com *hooks* basta implementar uma interface. Mesmo assim, o uso do ACF é mais simples que POA tradicional, pois dispensa a definição de pontos de corte no código customizado.

5.2.3. Robustez em relação à evolução do produto

As duas abordagens se mostraram robustas em relação à evolução do produto. Isto se deve ao fato de ambas se basearem no uso de interfaces para definir o contrato entre o código de personalização e o produto. No caso dos *hooks* eles são definidos implementando interfaces orientadas a objetos. No caso do ACF, as interfaces são definidas por meio de JPIs. Desde que as interfaces não mudem entre versões do produto, a compatibilidade das customizações é garantida.

Para o desenvolvedor do produto, é preciso maior disciplina quando se alteram métodos expostos pelo ACF. Se ele mudar a assinatura de um método, ele deve verificar se há uma anotação `@Exhibits` associada ao método. Se houver, ele precisa alterar a definição de ponto de corte incluída na anotação para mantê-la compatível com a nova assinatura. Esta verificação não é feita automaticamente pelo compilador Java.

Um ponto no qual o ACF se mostra menos robusto é quando customiza pontos de extensão não previstos no produto. Nesse caso, como os aspectos e classes adaptadoras não são construídos sobre interfaces JPI, eles sofrem da fragilidade dos pontos de corte. Mas não há comparação com a técnica de *hooks* neste caso, já que com *hooks* não é possível criar pontos de extensão não previstos.

5.2.4. Outras considerações

A programação orientada a aspectos é usada para reduzir o emaranhamento e o espalhamento de código. No item 5.2.2 foi descrito como o ACF é usado para reduzir o emaranhamento. Ele tem potencial também para reduzir o espalhamento, que é repetição do mesmo código em várias classes do sistema. Na prática, verifica-se que o espalhamento não é um problema relevante no contexto de customização. Nas propostas de personalização levantadas, não se encontrou nenhum caso em que um mesmo código de customização era aplicado em vários pontos de extensão simultaneamente. Todas as propostas de personalização pediam ajustes em pontos bem localizados.

Isto não significa que não possam ocorrer situações em que a repetição do mesmo ajuste em vários pontos seja necessária. Se isto ocorrer, o ACF poderá tratar esse caso, reduzindo o espalhamento. Mas os dados obtidos com o levantamento das propostas de customização é um indicador que isso não deve ser frequente.

Na abordagem baseada em *hooks*, é possível definir o tipo de *hook* que poderá ser implementado em um ponto de extensão. Por exemplo, o fabricante do produto pode definir que uma consulta poderá ser estendida, mas não substituída. Com JPIs, não há um mecanismo para restringir o tipo de *advice* (*before, after, around*) que será aplicado.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões deste trabalho e indicações de possíveis trabalhos futuros.

6.1. Conclusões

O *framework* ACF foi proposto como uma estratégia para usar programação orientada a aspectos na personalização de produtos de software. Ele aproveita uma das principais vantagens de POA, que é a de modularizar responsabilidades transversais, tratando a customização como se fosse uma responsabilidade transversal. Ao mesmo tempo, com o uso de JPIs, a ACF evita uma das principais deficiências de POA, que é a fragilidade dos pontos de corte.

Os testes comparativos entre o ACF e uma abordagem tradicional de customização baseada em *hooks* mostrou que o uso de aspectos como ferramenta de customização é vantajoso principalmente em softwares ainda não maduros. Nesses casos, o fabricante do produto ainda não conhece o suficiente quais são as necessidades de customização de seus clientes, e uma abordagem baseada em aspectos permite a criação de ajustes rápidos sem que o produto seja alterado.

Em produtos já maduros, nos quais a incerteza sobre os pontos de extensão é menor, a principal vantagem no uso de aspectos seria modularizar responsabilidades transversais de customização, o que elimina do código do produto os trechos de instrumentação para injeção de customizações.

6.2. Trabalhos Futuros

Neste trabalho a definição da notação @JPI seguiu de forma bem próxima a filosofia da JPI proposta por (Inostroza et al., 2011). As anotações `@Jpi` e `@Exhibits` definem as interfaces e pontos de corte praticamente da mesma forma que as palavras chave `jpi` e `exhibits` da JPI original. É possível aprimorar a notação @JPI criando notações adicionais que dispensem a definição de pontos de corte como expressões AspectJ, o que simplificaria o código. Também seria possível adicionar à notação a possibilidade de limitar os tipos de *advices* que uma JPI pode

aceitar, o que aumentaria o controle do fabricante do produto sobre o código a ser injetado.

(Bodden, 2011) estendeu a linguagem de JPIs para permitir a definição de *Closure Joinpoints*. O uso de *closure joinpoints* seria um facilitador para a definição de pontos de extensão, em particular para o caso de rotinas complexas, nas quais se deseja expor somente um trecho do código para customização. Pode ser interessante estudar uma forma de estender a notação @JPI para incorporar o conceito de *closure joinpoints*.

O *framework* ACF é somente um protótipo, e carece de várias ferramentas que facilitariam o trabalho de customizadores. Em particular, ferramentas e procedimentos para testes de *advices* criados para JPIs.

REFERÊNCIAS

ALUR, D.; CRUPI, J.; MALKS D. **Core J2EE Patterns: Best Practices and Design Strategies, Segunda Edição**. Prentice Hall/Sun Microsystems Press. 2003

BAGHERZADEH, M.; RAJAN, H.; LEAVENS, G. T.; MOONEY, S. **Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces.** Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11). ACM, New York, NY, EUA. 2011.

BOCKISCH, C.; MALAKUTI, S.; AKSIT, M., KATZ, S. **Making aspects natural: events and composition .** Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11). ACM, New York, NY, EUA, p. 285-300. 2011.

BODDEN, E. **Closure joinpoints: block joinpoints without surprises.** Proceedings of the tenth international conference on Aspect-oriented software development (AOSD '11). ACM, New York, NY, USA, 117-128. 2011

BREHM, L.; HEINZL, A.; MARKUS, M. **Tailoring ERP Systems: A Spectrum of Choices and their Implications.** Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS '01), Vol. 8. IEEE Computer Society, Washington, DC, USA, 8017. 2001.

FILMAN ,R. E.; FRIEDMAN D. P. **Aspect-oriented programming is quantification and obliviousness.** Technical Report 01.12, RICAS, 2001.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1995.

GASIUNAS, V.; SATABIN, L.; MEZINI, M.; NOYÉ, J.; NÚÑEZ, A. **Declarative events for object-oriented programming.** Technical Report 7313, INRIA, 2010.

GRADECKI, J. D.; LESIECKI, N. **Mastering AspectJ: Aspect-Oriented Programming in Java.** Wiley Publishing, 2003.

GRISWOLD, W. G.; SULLIVAN, K.; SONG, Y.; SHONLE, M.; TEWARI, N.; CAI, Y.; RAJAN, H. **Modular Software Design with Crosscutting Interfaces.** IEEE Software. Volume 23 Issue 1, 2006.

INOSTROZA, M.; TANTER, E.; AND BODDEN, E. **Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs**. Proceedings of the 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), New Ideas track, Szeged, Hungria, ACM Press. 2011.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.; IRVING, J. **Aspect Oriented Programming.** Proceedings of ECOOP. pp. 220-242, 1997.

KICZALES, G.; HILSDALE, E.; HUGUNIN,J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. **An Overview of AspectJ.** Proceedings of the 15th European Conference on Object-Oriented Programming. p. 327 - 353. Springer-Verlag. 2001.

KOPPEN, C.; STOERZER, M. **PCDiff: Attacking the Fragile Pointcut Problem.** European Interactive Workshop on Aspects in Software (EIWAS). 2004.

KULESZA, U., ALVES, V., GARCIA, A., LUCENA, C., BORBA, P. **Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming.** Proceedings of 9th International Conference on Software Reuse (ICSR'2006), Springer-Verlag, LNCS 4039, p. 231-245, Torino, Itália, 2006.

KULESZA, U.; COELHO, R.; ALVES, E.; NETO, A. C.; GARCIA, R.; LUCENA, C.; STAA, A. V., BORBA, P. **Implementing Framework Crosscutting Extensions with EJPs and AspectJ.** Proceedings of the ACM SIGSoft 20th Brazilian Symposium on Software Engineering. 2006.

LIGHT, B. **The maintenance implications of the customization of ERP software**. Journal of Software Maintenance and Evolution: Research and Practice. Volume 13, Issue 6, p 415–429, November/December 2001.

ROTHENBERGER, M.A. SRITE, M. **An Investigation of Customization in ERP System Implementations**.IEEE Transactions on Engineering Management, Nov. Volume: 56 Issue: 4 p. 663 - 676. 2009.

SESTOFT , P., VAUCOULEUR, S. **Technologies for Evolvable Software Products: The Conflict between Customizations and Evolution**. Advances in Software Engineering: Revised Tutorial Lectures, Springer-Verlag, Berlin, Heidelberg, 2008.

STEIMANN, F.; PAWLITZKI, T.; APEL, S.; KÄSTNER, C. **Types and Modularity for Implicit Invocation with Implicit Announcement.**ACM Transactions on Software Engineering and Methodology (TOSEM). 2009.

SULLIVAN, K.; GRISWOLD, W. G.; RAJAN, H.; SONG, Y.; CAI, Y.; SHONLE, M.; TEWARI, N. **Modular aspect-oriented design with XPIs.**ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 20 Issue 2, p. 5:1- 5:42. 2010.

VAUCOULEUR, S. **Customizable and upgradable enterprise systems without the crystal ball assumption.** Proceedings of the 13th IEEE international conference on Enterprise Distributed Object Computing (EDOC'09). IEEE Press, Piscataway, NJ, USA, 176-185. 2009

VAUCOULEUR, S. **Upgradeable Software Product Customization by Code Query.** Tese de Doutorado. IT University of Copenhagen. 2009.

ZACH, O.; MUNKVOLD, B. E. **ERP System Customization In SMEs: A Multiple Case Study.** PACIS 2011 Proceedings. Paper 221. 2011.

APÊNDICE A: EXEMPLOS DE USO DE HOOKS

Este apêndice apresenta exemplos de código fonte dos tipos de hooks descritos na seção 4.

a) Hooks de Substituição

São usados quando se deseja que um método customizado substitua um método padrão do produto. Nesse caso, só o método customizado é executado e o seu valor de retorno é usado no lugar do retorno do método padrão.

Hooks de substituição recebem os mesmos parâmetros que o método a ser substituído, e retornam valor do mesmo tipo.

```
01 /**
02  * Verifica se um aluno está matriculado regularmente
03  */
04 public boolean verificarAlunoRegular(Long idAluno)
05 {
06     boolean alunoRegular;
07     // Chama hook de substituição
08     if(hookSubstituicao!=null) {
09           RetornoHook<Boolean> retorno =
10                 hookSubstituicao.verificarAlunoRegular(idAluno);
11           if(retorno.isSubstitui())
12                 return retorno.getRetorno();
13     }
14
15     // Se não houver hook, segue a lógica padrão
16     // executando consultas de registros de matrícula do aluno
17     // e ajustando o valor da variável 'alunoRegular'
18     ...
19     return alunoRegular;
20 }
```

Listagem 19 - Exemplo de *hook* de Substituição

No exemplo da Listagem 19, a chamada ao *hook* de substituição ocorre no trecho das linhas 8 a 13. Primeiro é checado se foi definida uma implementação do *hook*. Caso positivo, ele é executado e retorna um objeto do tipo `RetornoHook` com duas propriedades, "substitui" e "retorno". A propriedade "retorno" guarda o valor de retorno do *hook*, e a propriedade "substitui" indica se este valor de retorno deve ou não ser usado como valor de retorno do método principal.

b) Hooks de Alteração

São usado quando se deseja que um método customizado altere o resultado de um método padrão do produto. O método padrão é executado primeiro. Ao final de sua execução é feita uma chamada ao *hook*, que pode ou não alterar o valor de retorno do método principal.

Hooks de alteração recebem os mesmos parâmetros que o método principal, mais um parâmetro adicional com o valor de retorno já calculado. O retorno do hook é usado como retorno do método principal. Na listagem 20, a chamada ao hook é feita nas linhas 17 a 19.

```
01 /**
02  * Retorna o número de aulas semanais de uma disciplina previstas para uma turma
03  */
04 public Integer obterAulasSemanais(Long idDisciplina,Long idTurma)
05 {
06     Integer aulasSemanais;
07
08     // Primeiro, executa consultas ao banco de dados para contar
09     // a quantidade de aulas por semana da turma informada
10     // preenchendo o valor da variável 'aulasSemanais'
11     //...
12
13
14     // Chama hook de alteração
15     // permitindo que o valor seja ajustado de acordo com critérios definidos
16     // pelo cliente
17     if(hookAlteracao!=null)
18           aulasSemanais = hookAlteracao.obterAulasSemanais(idDisciplina,
19                                                 idTurma, aulasSemanais);
20     return aulasSemanais;
21 }
```

Listagem 20 - Exemplo de *hook* de Alteração

c) Hooks Internos

São hooks injetado nos meio do código de um método. São usados para permitir a customização de passos intermediários de rotinas.

Na listagem 21, a chamada ao *hook* ocorre nas linhas 13 e 14. Nesse exemplo, a rotina a média final de um aluno em uma disciplina, e verifica se ela é suficiente para aprová-lo.

```
01 /**
02  * Verifica se o aluno foi aprovado em uma dada disciplina
03  * Calcula a média, arredonda o valor e verifica se
04  * está acima da nota mínima de aprovação
05  */
06  public Boolean verificarAprovacao(Long idAluno,Long idDisciplina,
07                                    Long idTurma) {
08     List<Nota> notas = obterNotas(idAluno,idDisciplina,idTurma);
09     Formula formulaMedia = obterFormulaCalculoMedia(idDisciplina,idTurma);
10     double notaAprovacao = obterNotaDeAprovacao(idDisciplina,idTurma);
11     double media = calcularMedia(formulaMedia,notas);
12
13     if(hookInterno!=null)
14           media=hookInterno.arredondaMedia(media,idDisciplina,idTurma);
15
16     if(media<notaAprovacao)
17           return false;
18     else
19           return true;
20 }
```

Listagem 21 - Exemplo de *hook* Interno

Ao contrário dos hooks de substituição e de alteração, os parâmetros e valor de retorno dos hooks internos não precisam ser os mesmos que o do método principal.

APÊNDICE B: EXEMPLOS DE OPERAÇÕES CUSTOMIZÁVEIS

Este apêndice apresenta exemplos de código-fonte das operações personalizadas por *hooks*, descritas na seção 4.

a) Função

Funções aqui são entendidas como rotinas que recebem um ou mais parâmetros e retornam um valor numérico ou caractere. São basicamente rotinas de cálculo ou totalização. Funções aceitam *hooks* de alteração e substituição.

```
public class FuncoesAcademicas {
	DatabaseUtil database;

	 ....

	/*
	 * Retorna a quantidade de créditos das disciplinas cursadas por um aluno
	 */
	public Number creditosCursados(Long idAluno)
	{
		//Hook de substituição
		if(hookSubstituicao!=null)
			return peSubstituicao.creditosCursados(idAluno);

		Number creditos=(Number)database.executeScalar(
				"select sum(creditos) from Matriculas
				where situacao='aprovado' and
				aluno=?",idAluno);

		//Hook de alteração
		if(hookAlteracao!=null)
			return peAlteracao.creditosCursados(creditos, idAluno);
		return creditos;
	}
}
```

Listagem 22 - Exemplo de Função (com *hooks*)

A listagem 22 mostra um exemplo da função `creditosCursados` que calcula o número de créditos que um aluno cursou em uma instituição acadêmica. Os *hooks* estão assinalados em vermelho.

b) Consulta

Consultas são métodos Java que fazem buscas em banco de dados e retornam listas de registros. Consultas também aceitam *hooks* de alteração e substituição.

```
public class ConsultasAluno {
	DatabaseUtil database;

	/*
	 * Lista as notas e faltas do aluno, por disciplina
	 */
	public List<Row> listarResumoHistorico(Long idAluno)
	{
		//Hook de substituição.
		if(hookSubstituicao!=null)
			return hookSubstituicao.listarResumoHistorico(idAluno);

		List<Row> lista=database.executeQuery(
				"select disciplina, media, faltas
				from historico where aluno=?", idAluno);

		//Hook de alteração.
		if(hookAlteracao!=null)
			return hookAlteracao.listarResumoHistorico(lista, idAluno);

		return lista;
	}
}
```

Listagem 23 - Exemplo de Consulta

A listagem 23 mostra o método `listarResumoHistorico` que traz do banco de dados uma lista de notas e faltas do aluno. Os *hooks* estão assinalados em vermelho.

c) Edição de Registro

As classes para manipulação de registros seguem o padrão de projeto *Data Access Object*, ou DAO (ALUR et al., 2003). A listagem 24 mostra um exemplo simplificado de uma classe que cadastra registros de alunos. O propósito da classe DAO é efetuar operações de inserção, alteração ou remoção de registros no banco de dados. Antes da operação em si, o registro é validado.

Quando um campo é adicionado em uma tela de cadastro, é preciso acrescentar também uma coluna extra na tabela correspondente no banco de dados, e uma propriedade a mais na entidade Java associada a essa tabela. Além disso, é preciso também alterar a classe DAO que manipula esse registro, para validar o campo novo e permitir que ele seja armazenado no banco de dados.

Consequentemente, as classes DAO são afetadas tanto na criação de novo campo em tela quanto na customização de operações de edição de registros.

```
public class AlunoDAO {
	DatabaseUtil database;
	HookAlunoDAO hook;

	public void inserir(Aluno aluno)
	{
		validarInsercao(aluno);
		if(hook!=null)
			hook.preInsercao(aluno);

		database.inserirEntidade(aluno);

		if(hook!=null)
			hook.posInsercao(aluno);
	}

	public void alterar(Aluno aluno)
	{
		validarAlteracao(aluno);
		if(hook!=null)
			hook.preAlteracao(aluno);

		database.alterarEntidade(aluno);

		if(hook!=null)
			hook.posAlteracao(aluno);
	}

	public void remover(Aluno aluno)
	{
		validarRemocao(aluno);
		if(hook!=null)
			hook.preRemocao(aluno);

		database.removerEntidade(aluno);

		if(hook!=null)
			hook.posRemocao(aluno);
	}
	public void validarInsercao(Aluno aluno)
	{
		//regras de validação aqui
		...
	}

	public void validarAlteracao(Aluno aluno)
	{
		//regras de validação aqui
		...
	}

	public void validarRemocao(Aluno aluno)
	{
		//regras de validação aqui
		...
	}
}
```

Hooks Internos

Listagem 24 - Exemplo de Edição de Registro

d) Rotinas genéricas

Rotinas que não se enquadram nas categorias de função, consulta ou edição de registro são classificadas como “genéricas”. Um caso típico de método que sofre customização é o de rotinas que executam operações relativamente complexas em uma sequência de passos, como mostrado na listagem 25.

```
public void trancarMatricula(Long idAluno, Date dataTrancamento)
{
        //Hook interno
        if(hook!=null)
                hook.validarTrancamento(aluno, dataTrancamento);

        trancarTurmas(idAluno,dataTrancamento);

        trancarHistorico(idAluno,dataTrancamento);

        //Hook interno
        if(hookCobranca!=null)
                hookCobranca.atualizarCobrancasPorTrancamento(aluno,
                                                        dataTrancamento);
        else
                atualizarCobrancasPorTrancamento(idAluno,dataTrancamento);

        gerarBoletosTrancamento(idAluno);

}
```

Listagem 25 - Exemplo de Rotina Genérica

Neste exemplo foram usados dois *hooks* internos. O primeiro faz uma validação opcional, ou seja, adiciona um passo extra no algoritmo. O segundo faz uma substituição de um passo intermediário.