# Adaptação e Manutenção de Sistemas Integrados de Gestão apoiados pela Programação Orientada a Aspectos

**Fernanda Almeida Passos**[1], **Kleber Tarcísio Oliveira Santos**[1], **Raphael Ferreira Santos Barreto**[1], **Alberto Costa Neto**[1]

[1] Departamento de Computação – Universidade Federal de Sergipe
Av. Marechal Rondon, Jardim Rosa Elze, 49100-000 – São Cristóvão – SE – Brasil

fernanda@ufs.br, klebertarcisio@yahoo.com.br

rfsbarreto@gmail.com, alberto@ufs.br

***Abstract.*** *Management Information Systems (MIS) usually demand product customizations for each client, frequently involving the creation of new components and changes to existing ones. As a result, MISs usually face maintenance and evolution problems. One of the major challenges is to identify the variations present on the current version and reapply them to newer versions of the system. This might demand rework if these variations are not properly separated from the base code. This paper presents a catalog of variations types found in a real MIS, which is being deployed at Federal University of Sergipe (UFS), and discuss how most common variations that occur in a MIS can be addressed by Aspect-Oriented Programming in a modular fashion.*

***Resumo.*** *Sistemas Integrados de Gestão (SIG) normalmente requerem a customização do produto para cada cliente, frequentemente envolvendo a criação de novos componentes e mudanças nos existentes. Por este motivo, SIGs normalmente enfrentam problemas de manutenção e evolução. Um dos maiores desafios é identificar as variações presentes na versão corrente e reaplicá-las nas versões mais novas do sistema. Isto pode demandar retrabalho se estas variações não estiverem adequadamente separadas do código base. Este artigo apresenta um catálogo de tipos de variações encontradas em um SIG real, que está sendo implantado na Universidade Federal de Sergipe (UFS), e discute como as variações que mais ocorrem no SIG são endereçadas pela Programação Orientada a Aspectos de forma modular.*

## 1. Introdução

A complexidade do software tem crescido cada vez mais tanto devido à necessidade de lidar com requisitos não funcionais complexos, como diferentes tipos de rede, protocolos, interfaces com usuário, sistemas operacionais, *Application Programming Interfaces (APIs)*, quanto pelas variações nos requisitos funcionais que são necessárias para que um software seja adaptado aos seus usuários finais e distribuidores. Uma das formas de lidar com isto é através da criação de Linhas de Produto de Software (Software Product Line — LPS) [Pohl et al. 2005, Rocha et al. 2012].

Uma LPS é um conjunto de sistemas intensos de software que compartilham um conjunto comum e gerenciado de características satisfazendo as necessidades específicas

de um segmento de mercado ou missão particular e que são desenvolvidos a partir de um conjunto comum de artefatos de uma forma prescrita [Clements and Northrop 2001]. Ao implantar adequadamente a cultura de LPS em uma empresa, torna-se mais fácil obter produtos com variações, sejam funcionais ou não funcionais, satisfazendo as necessidades dos clientes. Porém, é preciso conhecer as variações de cada produto e aplicar metodologias de desenvolvimento, processos, técnicas e ferramentas adequadas para tal fim.

Entretanto, nem sempre a empresa que desenvolveu uma LPS tem interesse em criar os artefatos necessários para os mais variados clientes, seja devido ao elevado número de clientes, ou por conta de questões econômicas e estratégicas. Isto leva a empresa que deseja utilizar uma LPS a desenvolver artefatos específicos para satisfazer seus requisitos funcionais e não funcionais, e ainda lidar com futuras evoluções na LPS, que normalmente geram impacto sobre estes artefatos. Outra questão é a alteração nos artefatos que fazem parte do núcleo da LPS para atender os requisitos do cliente. Isto implica em problemas futuros de manutenção e evolução, devido às novas versões da LPS terem que contemplar estes artefatos do núcleo readaptados.

Um exemplo desta situação pode ser encontrado em Sistemas Integrados de Gestão (SIG) — que são sistemas de informação que integram os dados e processos de uma organização em um único sistema [Laudon and Traver 2011] — como o SIG [SIG Software e Consultoria 2012] desenvolvido pela Universidade Federal do Rio Grande do Norte (UFRN) e atualmente em fase de implantação na Universidade Federal de Sergipe (UFS). O SIG foi construído usando a Programação Orientada a Objetos (POO) [Meyer 1997] e Padrões de Projeto [Gamma et al. 1995, Alur et al. 2003], mas as abordagens utilizadas atualmente pela referida equipe para customizar o sistema não estão se mostrando adequadas, devido à complexidade de identificar e reaplicar as customizações em novas versões, levando a atrasos na implantação de novos recursos e na manutenção dos que estão em produção. Tomando este cenário como referência, este artigo avalia a utilização da Programação Orientada a Aspectos (POA) [Kiczales et al. 1997] como uma possível solução para lidar com os problemas de manutenção e evolução que a equipe do Centro de Processamento de Dados (CPD) da UFS vem enfrentando para adaptar o SIG às necessidades desta instituição, conforme discutido em um trabalho anterior [Passos and Neto 2012]. É importante ressaltar que a abordagem proposta não consiste na refatoração do SIG para utilizar a POA, mas sim na aplicação desta última para adaptar o sistema sem que seja preciso modificar diretamente as classes existentes, ou seja, proporciona uma forma de modularizar as variações no sistema através de aspectos.

Desta forma, a Seção 2 descreve o SIG em mais detalhes e o processo de atualização das versões do SIG pela IFES UFS, relatando os problemas enfrentados durante a manutenção e evolução do mesmo. Na Seção 3 são identificados e discutidos os tipos de variações encontradas no código fonte de um módulo do sistema para algumas versões do SIG UFS. Logo em seguida, na Seção 4 são apresentadas propostas de utilização da POA para os tipos de variações mais frenquentemente encontradas no código fonte do SIG, analisando seus possíveis benefícios e fraquezas. Alguns trabalhos relacionados são brevemente descritos na Seção 5. Finalmente, a Seção 6 traz as conclusões e trabalhos futuros.

## 2. Sistema Integrado de Gestão

O SIG [SIG Software e Consultoria 2012] permite a gestão de todas as áreas de atuação das Instituições Federais de Ensino Superior (IFES), sendo composto de três grandes módulos: Atividades Acadêmicas (SIGAA), Recursos Humanos (SIGRH) e Patrimônio, Administração e Contratos (SIPAC).

As facilidades oferecidas pelo SIG despertaram o interesse das IFES, a exemplo da Universidade Federal da Bahia (UFBA), Universidade Federal do Maranhão (UFMA), Universidade Federal do Ceará (UFC), Universidade Federal do Pará (UFPA) e outras. A UFS, que atuava em um cenário ausente de interoperabilidade e de integração de sistemas, resolveu também firmar um acordo de parceria com a UFRN para aquisição do SIG, a fim de assegurar a geração de informações integradas e confiáveis a comunidade acadêmica e aos gestores da instituição. Diante desse novo cenário, passou então a ter acesso ao código fonte da aplicação, o qual recebe periodicamente atualizações visando corrigir problemas reportados e introduzir novos recursos.

A partir daí a equipe de desenvolvedores do CPD da UFS passou a efetuar as adaptações necessárias no SIG para atender às regras de negócio da UFS, já que estas regras diferem de uma IFES para outra. Como exemplo, podemos citar diferenças de cálculo da MGP dos alunos, prazos para matrícula/reformulação/inclusão/exclusão de disciplinas, existência ou não de provas finais, prioridades na distribuição de vagas nas disciplinas, além de melhorias específicas solicitadas por usuários da instituição como um filtro por período de cadastro em consultas de projetos de extensão, a modificação no texto do certificado de avaliação para os avaliadores de projetos de extensão submetidos ao PIBIX (Programa Institucional de Bolsas de Iniciação à Extensão) e dentre outras adaptações solicitadas pelos clientes da instituição.

Estas variações poderiam ser previstas pela equipe que desenvolve o SIG na UFRN, que poderia oferecer artefatos alternativos que contemplam estas variações, abordagem mais correta do ponto de vista de LPS. Porém, as variações nos requisitos são grandes e relativamente frequentes entre IFES, muitas vezes afetando vários pontos do sistema, tornando muito difícil o trabalho de gerenciar todas estas variações de forma centralizada e produzir artefatos alternativos ou até mesmo opcionais que satisfaçam a todas as IFES. Desta forma optou-se por uma abordagem de desenvolvimento descentralizado e independente, na qual cada IFES adquirente do SIG é responsável pela alteração e/ou criação de novas regras de negócio (artefatos) que possam ser necessárias.

Para um melhor entendimento da abordagem adotada atualmente pela IFES UFS, é mostrado o processo de atualização de versões do SIG na Figura 1. Inicialmente, ao ser adquirida uma versão do SIG, neste exemplo a versão 2.22 que foi a primeira versão implantada na UFS, são realizadas alterações no código fonte do SIG da UFS versão 2.22 para atender ao conjunto de regras de negócio existentes na instituição. Porém, ao ser disponibilizada uma versão mais nova do SIG pela UFRN é necessário um processo de integração das versões dos sistemas para obter a nova versão atualizada do SIG na UFS, ou seja, o processo de *merge* para as classes do sistema alteradas.

Para isso, é preciso fazer a reintrodução das adaptações feitas na versão 2.22 do SIG da UFS no código fonte atualizado do SIG da versão 2.23 da UFS. Esta atividade pode ser bastante complexa quando as adaptações realizadas na versão 2.22 forem feitas

em pontos do código fonte que sofreram alterações na versão 2.23, exigindo uma análise mais cuidadosa do código. Além disso, é provável que diante de uma nova versão, os usuários da UFS solicitem novas adaptações no SIG da UFS.

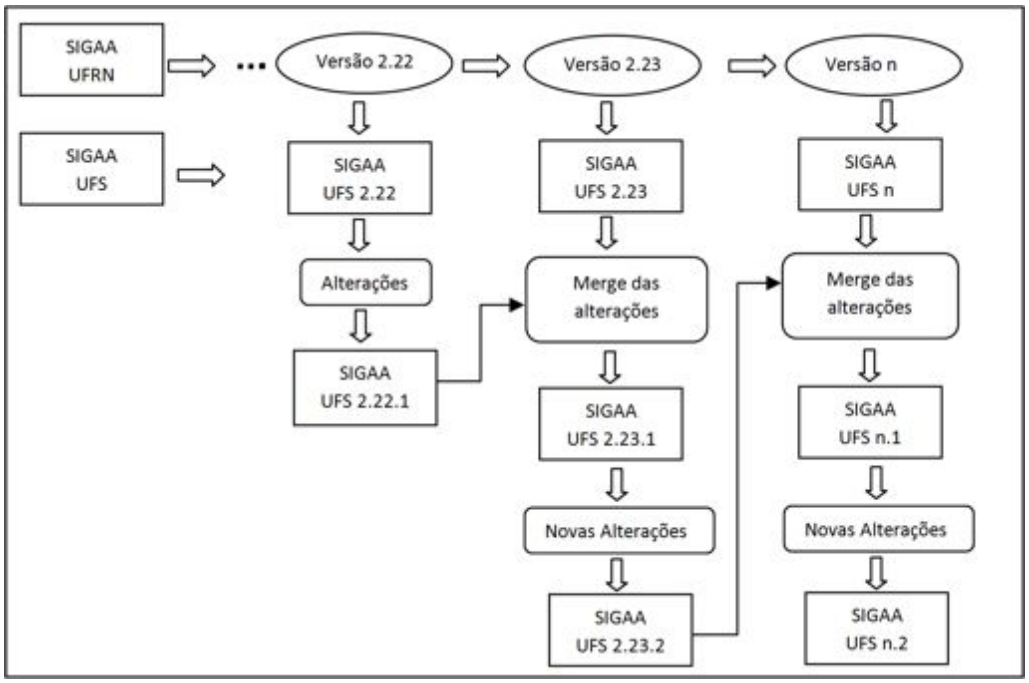


**Figura 1. Processo de atualização de versões do SIG.**

Sendo assim, a grande dificuldade que a equipe do CPD da UFS (e muito provavelmente todas as outras equipes das instituições que adquiriram o SIG) enfrenta é manter a sincronia do SIG adaptado para a UFS com a versão base da UFRN, pois a cada nova versão do SIG, é necessário reintroduzir as adaptações no código fonte, o que é bastante trabalhoso porque o mesmo evolui e exige que os desenvolvedores da UFS analisem novamente o código fonte. Este processo é tão demorado e sujeito a erros que muitas vezes ao ser concluído já deve ser reiniciado para a próxima versão.

## 3. Variações no SIG da UFS

Inicialmente foram realizados o levantamento e a catalogação dos tipos de variações no código fonte do SIG UFS, variações estas necessárias para atender às regras de negócio da UFS, considerando duas versões do SIG UFS correspondente a duas versões do SIG UFRN (2.22 e 2.23).

O processo de comparação das versões UFS e UFRN foi efetuado a partir da análise dos arquivos de extensão ".java" em que são implementadas as regras de negócios, as validações de atributos das classes domínio da aplicação e os acessos a base de dados (instruções SQL). Este processo foi realizado somente para o subsistema **Extensão** do módulo de **Atividades Acadêmicas (SIGAA)**, que tem como objetivo gerenciar as Ações de Extensão existentes na instituição. As ações de extensão são divididas em: Curso, Evento, Programa, Projeto e Produto. Para a comparação dos arquivos deste subsistema foi utilizado o aplicativo WinMerge [WinMerge 2013], que é uma ferramenta que permite

comparar duas pastas e arquivos, apresentando diferenças em um formato de texto com alguns recursos visuais.

As versões do SIGAA utilizadas na comparação foram a versão 2.22 do SIGAA da UFS que corresponde à versão 2.22 do SIGAA da UFRN e também a versão 2.23 de ambas as instituições. Cada alteração encontrada no código fonte da versão da UFS sobre a da UFRN foi classificada conforme um dos tipos descritos na Tabela 1.

**Tabela 1. Catálogo de tipos de variações encontrada no SIG UFS.**

| Sigla | Variação | Descrição da Variação |
|---|---|---|
| AC | Adicionar Classe | Adicionando uma nova classe ao projeto. |
| AM | Adicionar Método | Adicionando um novo método a classe. |
| AA | Adicionar Atributo | Adicionando um novo atributo a classe. |
| AV | Adicionar Variável local | Adicionando uma variável local a método. |
| APF | Adicionar parâmetro Formal | Adicionando um novo parâmetro formal a método |
| APR | Adicionar Parâmetro Real | Adicionando um novo parâmetro real a método. |
| ACM | Adicionar Chamada a Método | Adicionando uma nova chamada a método. |
| ACR | Adicionar Comando de Repetição | Adicionando um comando de repetição (for, while) a um método. |
| ACC | Adicionar Comando Condicional | Adicionando um comando condicional (if, switch) a um método. |
| MVA | Modificar Valor do Array | Modificando os valores de inicialização de um array definido como atributo estático da classe. |
| MVV | Modificar Valor de Variável local | Modificando o valor atribuído a uma variável local. |
| MPR | Modificar Parâmetro Real | Modificando o parâmetro real a método. |
| MCM | Modificar Chamada a Método | Modificando a chamada a método. |
| MER | Modificar Expressão condicional de Retorno | Modificando a expressão condicional da instrução de retorno (return), ou seja, alteração ou inserção de condições a expressão. |
| MLS | Modificar Literal String | Modificando a literal string a método que tem como tipo de retorno uma String. |
| MEC | Modificar Expressão Condicional | Modificando a expressão condicional (if, switch) a método, ou seja, alteração ou inserção de condições a expressão. |
| MIS | Modificar Instrução SQL | Modificando instrução SQL (select, insert, update, delete) definida em método. |
| MTM | Modificar Totalmente Método | Modificando totalmente a implementação do método. |
| EM | Excluir Método | Excluindo método. |
| EC | Excluir Código de método | Excluindo parte do código implementado a método. |

De posse do estudo inicial que envolveu o levantamento e a catalogação dos tipos de variações presentes nas classes das versões do sistema SIG UFS (2.22 e 2.23) foi possível reproduzir os gráficos da Figura 2 e Figura 3 com a identificação das alterações existentes juntamente com a quantificação das ocorrências de cada adaptação de acordo com os tipos das variações encontrados no código fonte das classes das versões do sistema escolhido para esse levantamento.

No gráfico da Figura 2 são mostrados os tipos de variações e as alterações existentes numeradas sequencialmente seguida do número de ocorrências de cada adaptação, por exemplo, para a variação "MPR" a mesma modificação "Alt 1" ocorre em três (3) pontos distintos dentre as classes da aplicação na versão 2.22 da UFS comparada com a mesma versão correspondente da UFRN de acordo com a classificação de variações da Tabela 1.

Vale ressaltar também que as alterações "Alt 1" da variação "MPR" e "Alt 1" da "MVA" e assim por diante são diferentes. Analisando o gráfico da figura 2 é detectado que as alterações "Alt 1" e "Alt 3" para o tipo de variação MPR são consideradas *crosscutting* devido ao número de ocorrências não ser unitário, ou seja, são adaptações idênticas que estão espalhadas e entrelaçadas em vários pontos das classes da aplicação, um bom indício de que a POA poderia ser útil para implementar tais alterações.

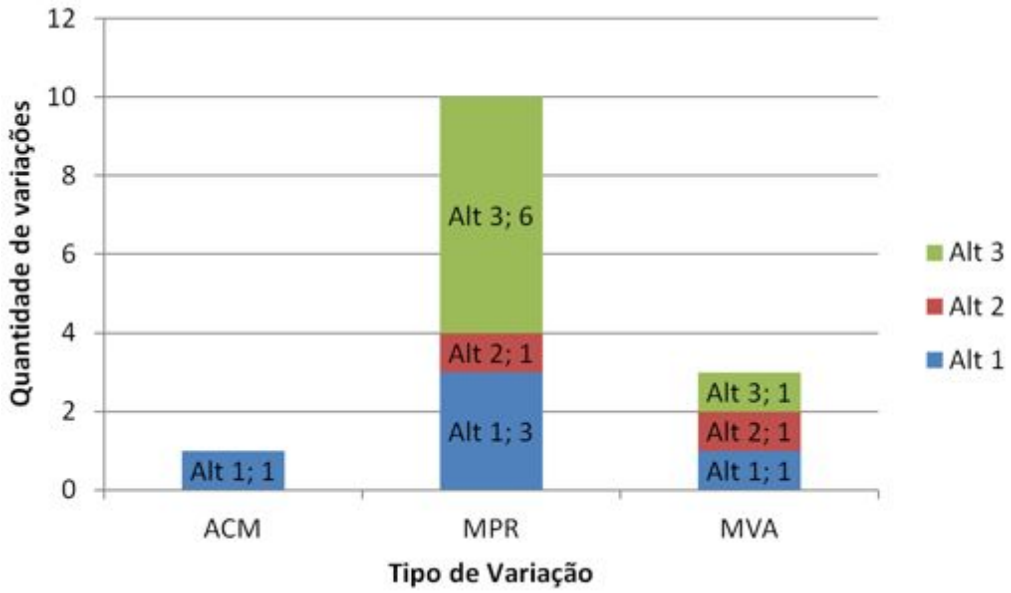


**Figura 2. Quantificação das ocorrências dos tipos de variações na versão 2.22 do SIG UFS.**

No gráfico da Figura 3 são catalogadas mais alterações juntamente com o número de ocorrências referentes a cada tipo de variação encontrada na aplicação de acordo com a classificação da Tabela 1 para a versão 2.23 da UFS, comparada com a versão correspondente da UFRN. Como podem ser observadas, outras adaptações foram identificadas e classificadas conforme os tipos das variações, sendo um reflexo de novos requisitos da UFS. Observa-se também no gráfico que há alterações (adaptações) cujo número de ocorrências não é unitário, ou seja, são alterações idênticas que se repetem em vários pontos do código fonte da aplicação, identificadas como crosscutting por estarem espalhadas e entrelaçadas no código. Isto ocorre principalmente para os tipos de variação ACC, ACM, AV, MPR identificados no código fonte da aplicação o que nos leva a crer que podem ser bem resolvidas utilizando a POA. Analisando o gráfico é perceptível que o tipo de variação MPR (modificação de parâmetro real) é o mais frequente e possui várias alterações que se caracterizam como *crosscutting*. De um total de dezesseis (16) adaptações diferentes, sete (7) delas (“Alt 9”, “Alt 10” e “Alt 13” com duas (2) ocorrências, “Alt 1” e “Alt 14” com três (3) ocorrências, “Alt 3” com seis (6) ocorrências, “Alt 12” com vinte e uma (21) ocorrências totalizando trinta e nove (39) ocorrências que estão espalhadas e entrelaçadas.

Portanto, o cenário atual baseado nesse levantamento, reforça a utilização da POA para efetuar a modularização dessas adaptações, sendo possível interceptar vários pontos de interesse no código fonte das classes alteradas com a finalidade de inserir um novo comportamento correspondente à adaptação requerida.

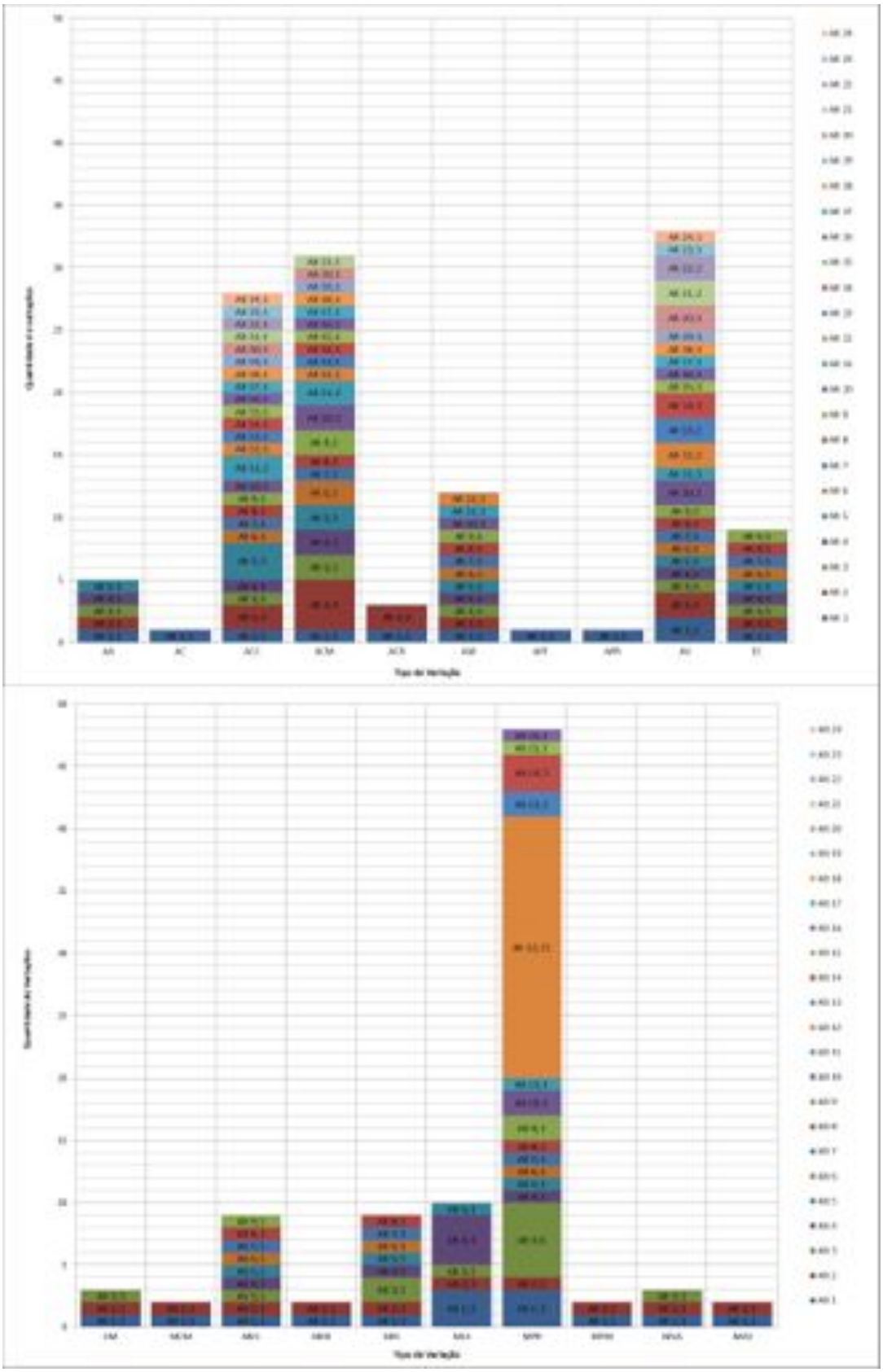

**Figura 3. Quantificação das ocorrências dos tipos de variações na versão 2.23 do SIG UFS.**

## 4. Como implementar as variações catalogadas utilizando a POA

Considerando os problemas descritos na Seção 2, a seguir são apresentadas em linhas gerais formas de implementar os tipos de variações no código, catalogados na Seção 3 e constantes na Tabela 1, através de aspectos codificados em AspectJ [Kiczales et al. 2001], focando principalmente nas variações com características transversais (*crosscutting*). Estas variações (adaptações) são necessárias sempre que novas versões do SIG são liberadas e idealmente não devem demorar a ser efetuadas porque podem envolver questões de segurança ou correções urgentes. Por limitação de espaço, só serão discutidos neste artigo os tipos de variações mais frequentes.

A POA permite definir, além de classes e interfaces, um novo elemento chamado aspecto. Idealmente, um aspecto deve concentrar conceitos que normalmente estão espalhados pelo código e entrelaçados com outros conceitos, não sendo possíveis de serem implementados de forma localizada e modular em uma classe. Exemplos destes conceitos são, dentre outros: *logging*, *tracing*, distribuição e autorização. AspectJ permite definir em que partes do código um certo comportamento deve ser executado. A forma oferecida para definir o local é através de uma declaração de *pointcut*, dentre os quais pode-se destacar o *pointcut call*, que provê a capacidade de capturar chamadas de vários métodos em partes diferentes do código. Tendo definido o local onde um comportamento deve ser inserido, utiliza-se um *advice* associado ao pointcut escolhido para definir o comportamento que deve ser executado naquele(s) ponto(s) do código. Em resumo, a POA permite modularizar *crosscutting concerns* (conceitos transversais) através da declaração de aspectos contendo declarações de *pointcut* e *advice*, evitando a repetição e espalhamento de código pelo sistema.

A estratégia adotada procura manter o código base fornecido pela UFRN intacto e implementar aspectos que interceptam os pontos de interesse no código (*join points*) através de *pointcuts*, que especificam o conjunto de *join points* a serem interceptados, e utilizando-se de um *advice* define o comportamento (de forma similar a um método) a ser alterado na ocorrência dos *join points*. Desta forma, é possível modularizar as adaptações no aspecto. Logo, estariam separadas do código base do SIG UFRN, o que facilita a visualização do que fora modificado e ainda cria uma oportunidade para lidar melhor com as atualizações no SIG UFS em decorrência de novas versões do SIG UFRN.

Para o tipo de variação de maior frequência nas alterações realizadas no código fonte do SIG, a **MPR (modificação de parâmetro real)**, o objetivo geral é fazer com que um valor diferente seja passado em uma chamada de método. Sendo assim, uma das formas de fazer isso com aspectos é através da definição de um *pointcut* responsável por interceptar o *join point* de chamada ao método (usando o *pointcut designator call*) e fazer a ligação dos parâmetros reais originais com parâmetros do *pointcut* (utilizando o *pointcut designator args*). Estes parâmetros estarão disponíveis no *advice* do tipo *around* e poderão ser usados para realizar a chamada com os parâmetros reais desejados de dentro do *advice*, tarefa esta que envolve a chamada ao comando *proceed* que retoma a chamada interrompida com os parâmetros modificados.

No caso da variação **ACM (adicionar chamada a método)**, necessita-se adicionar uma nova chamada de método, podendo ocorrer em três pontos dentro de um método: no inicio, no meio ou no final. Quando a chamada precisa ser adicionada no início ou no final de um método, sugere-se declarar um *pointcut* que intercepte a chamada a este mé-

**Tabela 2. Sugestões de como implementar as variações com aspectos.**

| Variação | Local de ocorrência | Implementação com aspectos |
|---|---|---|
| MPR | Início | pointcut + args + advice around + proceed |
| | Meio | |
| | Fim | |
| ACM e ACC | Início | pointcut + advice before |
| | Meio | pointcut + advice around + proceed |
| | Fim | pointcut + advice after |
| AV | Início/Meio/Fim | Variável local em advice |

todo e adicione a chamada através de um *advice* do tipo *before* e *after*, respectivamente, os quais efetuam a chamada ao método que precisa ser adicionada. Já no caso da chamada ter que ser adicionada no meio do método, é preciso selecionar um *join point* ao redor do local onde a chamada seria adicionada e especificar um *pointcut* capaz de interceptá-lo. Em seguida, cria-se um *advice* do tipo *around* que executa a chamada de método seguido de um proceed para que o *join point* interceptado seja executado. Porém, para esse último caso, existe uma forte dependência com relação à implementação do método, o que pode levar este *join point* a sumir ou até outros serem introduzidos dentro do mesmo método, levando, respectivamente, à não realização da chamada de método e à execução duplicada da chamada de método.

Para o tipo de variação **ACC (adicionar comando condicional)** que representa adicionar um comando condicional a método é sugerido o tratamento da variação como descrito para a variação ACM ressaltando que todo o código fonte contido dentro do comando condicional será implementado no aspecto.

Finalmente, a variação **AV (adicionar variável local)**, que consiste na inclusão de uma variável local em um método, não é suportada por construções específicas da POA, mas ainda assim foi possível implementar esta variação com aspectos, bastando partir do princípio de que se uma variável nova foi criada, apenas um trecho de código que foi incluído (ou modificado) por outra alteração pode fazer uso da mesma. Como este trecho de código também deve ser transferido para um advice contido em um aspecto, é natural que esta variável também seja transferida para este aspecto.

A Tabela 2 aponta sucintamente possíveis formas de implementar, com recursos disponíveis na POA, os tipos de variações mais frequentes, discutidas nos parágrafos anteriores, segundo o levantamento inicial feito neste artigo das alterações de código nas versões do sistema SIG UFS (2.22 e 2.23). É importante ressaltar que todas as variações encontradas e contabilizadas nos gráficos das Figuras 2 e 3 puderam ser convertidas em aspectos com comportamento equivalente.

No entanto, é importante relatar que existem alguns riscos associados à adoção da POA, como o código fonte da UFRN mudar de uma forma que o aspecto deixe de funcionar conforme desejado, seja pelo desaparecimento de um *join point* ou pelo surgimento inesperado de um *join point*. Sendo assim, é importante enfatizar as dependências que existem entre aspectos e classes base do SIG, que se não forem respeitadas podem fazer com que o aspecto não tenha o comportamento programado. Para isso, os desenvolvedores da UFS devem estabelecer e checar condições que o código base deve atender.

## 5. Trabalhos Relacionados

Além da POA, existem outras técnicas para lidar com variações em LPS baseadas em transformação de código, dentre as mais conhecidas, estão as abordagens anotativas e composicionais, sendo que cada uma delas tem um nível de granularidade diferente, ou seja, atendem a variações que partem desde o grão mais fino (nível léxico), como a compilação condicional [Kästner and Apel 2009], a variações de componentes inteiros, como componentes distribuídos [Alur et al. 2003].

Em outros trabalhos [Alves 2007], boa parte destas técnicas foi comparada, no contexto de LPS, com a POA o que motivou sua escolha para a implementação dos tipos de variações mais frequentes catalogados neste trabalho para as adaptações existentes no SIG UFS. Porém, o problema abordado aqui é diferente, já que essas variações são específicas de cada IFES adquirentes e não fazem parte do SIG.

CIDE [Kästner and Apel 2009] é uma ferramenta pertencente à categoria das técnicas anotativas. Surgiu como uma evolução das ferramentas de pré-processamento, substituindo as marcações no código por cores, sendo que cada cor representa uma das *features* do software. O produto final é gerado de acordo com as *features* que ele deverá conter. Apesar de ter sido confirmado que o CIDE lidaria bem com todas as variações catalogadas (devido ao seu nível granularidade), não se teria ganho substancial em relação ao processo que já vem sendo feito, pois uma vez que uma nova versão do SIG fosse lançada, todos os locais que sofreram variações teriam que ser remarcados.

JaTS [Castor and Borba 2001] é uma ferramenta de transformação de código baseada em Java que se encaixa na abordagem das técnicas composicionais. Essa ferramenta se utiliza de *templates* de correspondência de entrada e saída. Os *templates* de entrada são descritos com base nos padrões da classe que se deseja modificar, enquanto que os de saída devem descrever as modificações que serão realizadas. Uma vez descritos os *templates*, passam-se como parâmetros para o JaTS as classes que devem ser analisadas, se alguma classe do conjunto passado casar com o padrão descrito nos *templates* de entrada ela será transformada gerando uma classe de acordo com o padrão descrito no *template* de saída. Esta ferramenta, assim como a POA, nos permite aplicar as variações sobre a versão base do SIG sem que seja necessário alterar diretamente o código base, refletindo-se em uma vantagem durante a evolução do sistema.

Feature-Oriented Programming (FOP) é um abordagem na qual as *features* são encapsuladas como incrementos sobre um programa base existente, juntamente com um mecanismo para combinar diferentes *features* sob demanda. Apesar de FOP e POA suportarem a implementação de interesses transversais heterogêneos, quando se trata de interesses transversais homogêneos, o recurso de quantificação presente na POA permite aplicar um incremento do mesmo comportamento sobre o programa base em vários pontos de forma localizada, enquanto que com FOP é necessário ter vários incrementos iguais aplicados em vários pontos [Lopez-Herrejon et al. 2006].

Este trabalho, possui uma contribuição com relação a outros trabalhos [Kästner and Apel 2009], [Lopez-Herrejon et al. 2006], por permitir a aplicação das variações sobre a versão base do SIG sem que seja necessário alterar diretamente o código base, refletindo-se em uma vantagem durante a evolução do sistema. Além disso, diferentemente de outros trabalhos [Alves 2007], a proposta deste trabalho é aplicar a POA

para adaptar o sistema sem que seja preciso modificar ou refatorar as classes existentes, através da modularização das variações do sistema em aspectos.

## 6. Conclusão e Trabalhos Futuros

Este artigo apresenta alguns problemas enfrentados durante a manutenção e evolução da versão customizada do SIG para a UFS, descrevendo os tipos de variações realizadas pela equipe do CPD da UFS para satisfazer os requisitos dos usuários locais para a implantação de tal sistema. Estas variações foram catalogadas para melhor entendimento dos tipos de alterações necessárias. Como base nas variações encontradas em duas (2) versões customizadas do sistema, foram implementadas versões equivalentes utilizando a POA, sendo que todas foram implementadas com sucesso sem efetuar qualquer alteração no código original do SIG fornecido pela UFRN.

Considerando a quantidade de tipos de variações implementadas com aspectos, o trabalho aponta para bons resultados com relação à separação do código fonte produzido pela equipe da UFS do original fornecido pela UFRN. Entretanto, como o estudo envolveu apenas um módulo do sistema, não é possível afirmar que outros módulos também se beneficiariam desta abordagem. Avaliar outros módulos é um dos trabalhos futuros.

De certa forma, esta abordagem não é inovadora no sentido de que trabalhos anteriores já discutiram a viabilidade de implementar variações em LPS com aspectos [Alves 2007, Kästner and Apel 2009], mas a proposta difere pelo fato de a UFRN não adotar uma abordagem baseada em LPS. Isto faz com que os riscos sobre a manutenção e evolução dos aspectos sejam maiores que em uma LPS que usa aspectos para implementar as variações porque as alterações no código base podem ser efetuadas de forma concorrente com os aspectos. No caso da abordagem proposta, apenas após receber o código base atualizado é que a equipe da UFS faria as adaptações (ou adequações dos aspectos existentes).

Como trabalhos futuros, pretende-se investigar alguma abordagem para especificar as dependências existentes entre classes e aspectos, preferencialmente alguma que proveja checagem automatizada. Finalmente, não estão descartadas outras técnicas de transformação de código, como algumas estudadas em trabalhos anteriores [Santos et al. 2013] que são muito utilizadas no contexto de LPS, sendo um dos trabalhos futuros avaliar quais seriam viáveis para lidar com as variações catalogadas.

É importante ressaltar que o problema aqui abordado é uma realidade em outras Instituições, podendo estar presente em qualquer situação que envolva a criação de um software baseado em um código fonte já existente, mas que evolui paralelamente. Um outro exemplo real é a customização de plataformas de software, como Android [Android 2012], para satisfazer aos requisitos dos fabricantes de hardware.

## Agradecimentos

Gostaríamos de agradecer ao CNPq, CAPES e Fapitec, agências brasileiras de fomento à pesquisa, por apoiar parcialmente este trabalho.

## Referências

Alur, D., Crupi, J., and Malks, D. (2003). *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall Ptr.

Alves, V. (2007). *Implementing Software Product Line Adoption Strategies*. PhD thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil.

Android (2012). Android Developers. http://developer.android.com, Maio.

Castor, F. and Borba, P. (2001). A Language for Specifying Java Transformations. In *Proceedins of V Brazilian Symposium on Programming Languages (SBLP 2001)*, pages 236–251.

Clements, P. and Northrop, L. M. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Kästner, C. and Apel, S. (2009). Virtual separation of concerns – a second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6).

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*.

Laudon, K. C. and Traver, C. G. (2011). *Management Information Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 12th edition.

Lopez-Herrejon, R., Batory, D., and Lengauer, C. (2006). A Disciplined Approach to Aspect Composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA. ACM.

Meyer, B. (1997). *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Passos, F. A. and Neto, A. C. (2012). Aplicando a Programação Orientada a Aspectos na Melhoria do Processo de Adaptação e Manutenção de Sistemas Integrados de Gestão. In *II Semana de Informática da UFS/Itabaiana)*.

Pohl, K., Günter, B., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Heidelberg, Germany.

Rocha, R., Fantinato, M., and Barros, V. A. (2012). Contribuições de Linha de Produto e Orientação a Serviços no Desenvolvimento de Sistemas de Informação. In *VIII Simpósio Brasileiro de Sistema de Informação*.

Santos, K. T. O., Barreto, R. F. S., Passos, F. A., and Neto, A. C. (2013). Utilizando JaTS e XVCL para customização em Sistemas Integrados de Gestão. In *Erbase 2013 - XIII Escola Regional de Computação Bahia Alagoas Sergipe*.

SIG Software e Consultoria (2012). Sistema Integrado de Gestão. http://www.sigsoftware.com.br, Maio.

WinMerge (2013). WinMerge. http://www.winmerge.org, Maio.