UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Estudo de Caso da Adoção da Programação Orientada a Aspectos para Melhoria do Processo de Manutenção e Evolução de Sistemas Integrados de Gestão

Lidiany Cerqueira Santos

SÃO CRISTÓVÃO/ SE

2015

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lidiany Cerqueira Santos

Um Estudo de Caso da Adoção da Programação Orientada a Aspectos para Melhoria do Processo de Manutenção e Evolução de Sistemas Integrados de Gestão

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alberto Costa Neto

SÃO CRISTÓVÃO/ SE

2015

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL
UNIVERSIDADE FEDERAL DE SERGIPE

Santos, Lidiany Cerqueira
S237u Um estudo de caso da adoção da programação orientada a aspectos para melhoria do processo de manutenção e evolução de sistemas integrados de gestão / Lidiany Cerqueira Santos ; orientador Alberto Costa Neto. – São Cristóvão, 2015.
133 f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Sergipe, 2015.

1. Programação Orientada a aspecto. 2. Software - Otimização. 3. Sistemas de informação gerencial. I. Costa Neto, Alberto, orient. II. Título.

CDU 004.432.2

Lidiany Cerqueira Santos

Um Estudo de Caso da Adoção da Programação Orientada a Aspectos para Melhoria do Processo de Manutenção e Evolução de Sistemas Integrados de Gestão

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Alberto Costa Neto, Presidente

Universidade Federal de Sergipe (UFS)

Prof. Dr. Márcio de Medeiros Ribeiro, Membro

Universidade Federal de Alagoas (UFAL)

Prof. Dr. Michel dos Santos Soares, Membro

Universidade Federal de Sergipe (UFS)

Um Estudo de Caso da Adoção da Programação Orientada a Aspectos para Melhoria do Processo de Manutenção e Evolução de Sistemas Integrados de Gestão

Este exemplar corresponde à redação final da Dissertação de Mestrado, sendo o Exame de Defesa da mestranda **Lidiany Cerqueira Santos** para ser aprovada pela Banca examinadora.

São Cristóvão - SE, 31 de Agosto de 2015

_____

Prof. Dr. Alberto Costa Neto

Orientador

_____

Prof. Dr. Márcio de Medeiros Ribeiro

Membro

_____

Prof. Dr. Michel dos Santos Soares

Membro

Dedicatória

Para Leandro.

Agradecimentos

Há tantas pessoas maravilhosas para agradecer pela conclusão dessa etapa na minha vida que certamente seriam necessárias mais do que algumas linhas para inserir o nome de todas aqui. Mas aproveitarei o espaço para agradecer a algumas pessoas em especial: Em primeiro lugar, gostaria de agradecer à Leandro, que me ajudou, apoiou incondicionalmente ao longo desses dez anos em que nos conhecemos e foi imprescindível para a conclusão desse curso. À minha mãe, Lucidalva, por ser minha maior fonte de inspiração, pelo carinho, apoio e amor necessário que me trouxeram até aqui. Sem ela, certamente tudo isso teria sido impossível. Ao meu orientador, professor Alberto, pela paciência, dedicação e participação fundamental para apresentação desse trabalho, pelos ensinamentos e conhecimento transmitidos ao longo desses trinta meses de convivência, que muito contribuíram para minha formação como estudante, profissional e pesquisadora. À todos os colegas de trabalho, pelo suporte para concluir o curso e realizar minhas atividades, principalmente à Fernanda, Matheus, Leonardo, e Luiz, que possibilitaram a realização desse estudo de caso na UFS. À amiga Juliana Lobo pelo incentivo para continuar quando pensava em desistir e pelo ombro amigo nas horas em que precisei. À Thayssa pela amizade, pelo apoio e pelas vírgulas e pronomes colocados nos seus devidos lugares. Aos amigos da *Sala do Conselho* por segurarem as pontas na minha ausência. À todos os docentes do PROCC, pela contribuição na minha formação e por possibilitarem a realização desse curso. Também aos colegas discentes, principalmente à Franklin, Sávio e Diego, pelos trabalhos realizados em conjunto e pela amizade. Enfim, tenho muito a agradecer à todos os amigos, amigas e familiares que me apoiaram e tornaram a realização desse sonho possível.

"Peço-lhes que escrevam todo tipo de livros, não hesitando diante de nenhum assunto, por mais banal ou mais vasto que sejam."

VIRGÍNIA WOOLF

Resumo

A necessidade de atender a diferentes clientes e incluir diferentes requisitos aumentam a complexidade da manutenção e evolução de sistemas, envolvendo tarefas de customização e adaptação para corrigir problemas e incluir novas funcionalidades. Atualmente a equipe de desenvolvimento da Universidade Federal de Sergipe vem trabalhando com a customização e manutenção de Sistemas Integrados de Gestão que informatizam operações fundamentais para a gestão acadêmica e administrativa. Para atender aos requisitos da universidade, são necessárias modificações constantes no código fonte original do sistema, isso demandou a criação de um processo adotado atualmente para gerenciar essa atividade. No entanto, a abordagem atual adotada pela UFS não vem se mostrando adequada, pois as adaptações introduzidas pela equipe de desenvolvimento estão espalhadas e entrelaçadas com o código original, dificultando as tarefas de identificar e reaplicar as customizações em novas versões, causando atrasos na implantação de novos recursos e na manutenção dos que estão em produção. Nesta pesquisa é apresentado um estudo de caso que avalia a adoção da Programação Orientada a Aspectos na adaptação, manutenção e evolução de sistemas customizados em comparação ao processo adotado atualmente na UFS. Como resultado, observou-se que foi possível implementar 99,68% dos tipos de variações utilizando a POA e um pequeno número de erros foi detectado após a reintrodução das variações com a POA. Contudo, esses erros precisam ser avaliados cuidadosamente devido à ocorrência de conflitos de versão. Os resultados obtidos com a avaliação do estudo indicam que há benefícios com a adoção da POA, porém os desenvolvedores confirmaram a existência de alguns obstáculos que precisam ser mitigados para adoção da POA no contexto avaliado.

Palavras-chave: Programação orientada a Aspectos, customização de software, variabilidade, Sistema Integrado de Gestão

Abstract

The complexity of maintenance and evolution of systems is increased whenever it is necessary to meet different customers and to include diverse requirements, involving customization and adaptation tasks to fix problems and add new features. Currently the development team of the Federal University of Sergipe is working with the customization and maintenance of Integrated Management Systems that automate fundamental operations for the academic and administrative management. Constant changes in the original system source code to meet the requirements of the university are needed. Because of that, it was required the creation and adoption of a process to manage this activity. However, this current approach is not showing to be adequate, since the changes made by the development team are tangled and also crosscutting the original code, complicating the identification and reaplication of the customizations in newer system versions, causing delays in the implementation of new features and maintenance of which are in production. This research presents a case study that evaluates the adoption of Aspect-Oriented Programming in adaptation, maintenance and evolution of customized systems in comparison to the process currently adopted by UFS. As a result, it was observed that it was possible to implement 99.68% of the types of variations using AOP, and a small number of errors were detected after the module update using AOP. However, these errors need to be carefully evaluated due to the occurrence of version conflicts. The results of the study indicate that there are benefits with the adoption of the AOP, but developers confirmed the existence of some obstacles that need to be addressed before the adoption of the POA in this cenario.

Keywords: Aspects-oriented programming, Software customization, Variability, Integrated management system

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AJDT - *AspectJ Development Tools*

APF - Administração Pública Federal

CPD - Centro de Processamento de Dados

IDE - *Integrated Development Environment*

JEE - *Java Enterprise Edition*

LPS - Linha de Produtos de Software

POA - Programação Orientada a Aspectos

POO - Programação Orientada a Objetos

SGBDOR - Sistema Gerenciador de Banco de Dados Objeto Relacional

SIG - Sistema Integrado de Gestão

SO - Sistema Operacional

UFRN - Universidade Federal do Rio Grande do Norte

UFS - Universidade Federal de Sergipe

Capítulo 1

Introdução

A maioria dos programas são modificados e alterados constantemente ao longo do seu ciclo de vida, por conta das mudanças de requisitos, o código-fonte é modificado continuamente (ORAM; WILSON, 2007). Alterações e adaptações constantes podem impactar negativamente sobre a qualidade do software, quando não são tratadas com a importância necessária (MENS *et al.*, 2005).

A manutenção de sistemas é uma necessidade de organizações como a Universidade Federal de Sergipe que adiquirem sistemas prontos e necessitam customizá-los. São realizadas mudanças nos sistemas adquiridos com objetivo de atender a novos requisitos funcionais e não funcionais que são necessários em uma organização. Para realizar essas modificações, essas organizações precisam compreender os sistemas originais adquiridos e desenvolver melhorias e adaptações de forma a contemplar requisitos funcionais e não funcionais que atendam às suas demandas, além de tratar evoluções de versões desses sistemas que possam impactar as melhorias desenvolvidas.

Esta situação ocorre atualmente com os Sistemas Integrados de Gestão (SIG), sistemas de informação para integração de dados e processos (LAUDON; TRAVER, 2011), desenvolvido pela Universidade Federal do Rio Grande do Norte (UFRN) e que vem sendo adotado por diversos órgãos da administração pública federal. A Universidade Federal de Sergipe (UFS) foi um dos órgãos que implantou o SIG [1] e que continua realizando o processo de

[1] `http://info.ufrn.br/wikisistemas/`

customização e adaptação do sistema para atender as regras de negócio da instituição, já que estas regras mudam após alterações na legislação e no regimento interno da instituição. O SIG foi desenvolvido utilizando-se a Programação Orientada a Objetos (POO) (MEYER, 1997) e Padrões de Projeto (GAMMA *et al.*, 1995; ALUR; CRUPI; MALKS, 2003). Porém, conforme estudo realizado por Passos e Neto (2012), as abordagens utilizadas atualmente pela equipe de desenvolvimento para customizar o sistema não estão se mostrando adequadas, devido à complexidade para identificar e reaplicar as customizações em novas versões, causando atrasos na implantação de novos recursos e na manutenção dos que estão em produção.

Na abordagem adotada hoje pela UFS para customização do SIG, as adaptações introduzidas pela equipe de desenvolvimento estão espalhadas e entrelaçadas, criando dificuldade para modularizar as adaptações em uma unidade de implementação (PASSOS; NETO, 2012). Além disso, a equipe enfrenta grande dificuldade para manter as versões do SIG adaptado para a UFS em sincronia com a versão base da UFRN. A cada nova versão do SIG liberada pela UFRN, é necessário reintroduzir todas as adaptações realizadas pela UFS no código fonte (PASSOS; NETO, 2012). Da forma como é realizado atualmente, esse processo é bastante custoso para a equipe, além de sujeito a erros, pois o sistema evolui, exigindo que os desenvolvedores da UFS analisem novamente todo o código fonte. Muitas vezes, ao concluir esse processo, uma nova versão já está disponível e o processo deve ser reiniciado. Esta dificuldade é consequência da forma como são realizadas as adaptações pelos desenvolvedores da UFS sobre o código fonte fornecido pela UFRN.

Diante das dificuldades citadas e da importância da adaptação e manutenção de sistemas, Passos *et al.* (2014) realizou um estudo com a finalidade de avaliar abordagens que possibilitem a implementação de variações nos requisitos das organizações sem alterar diretamente o sistema base adquirido, apoiando, ainda, sua evolução. O trabalho de Passos *et al.* (2014) reconhece que o processo de adaptação e manutenção de sistemas originais adquiridos ocorre frequentemente e que as atividades envolvidas nele não são triviais, muitas vezes afetando diversos pontos do sistema base e dificultando o gerenciamento das variações de forma modular. Outra dificuldade consiste em tratar novas versões do sistema base, realizadas pelos desenvolvedores, que podem gerar impacto sobre as variações desenvolvi-

das posteriormente. Desse modo, evidenciado que esse processo necessita de técnicas para um gerenciamento adequado das variações e da evolução dos sistemas. Uma proposta para lidar com essas dificuldades é a abordagem de Linhas de Produtos de Software (LPS). Seu principal benefício é facilitar o desenvolvimento de sistemas com variações (funcionais e não funcionais), satisfazendo necessidades de diferentes clientes. Uma Linha de Produtos de Software é um conjunto de sistemas de software que compartilham especificações para atender a um mercado ou propósito particular e desenvolvidos a partir de uma base em comum (POHL; BÖCKLE; LINDEN, 2005). A adoção de uma LPS reconhecidamente proporciona melhorias na qualidade, no tempo e no esforço de desenvolvimento, no custo e na complexidade da criação e da manutenção de sistemas (CLEMENTS; NORTHROP, 2001).

No trabalho conduzido por Passos *et al.* (2014), foram avaliadas técnicas que lidam com variações em Linhas de Produtos de Software, visando o aprimoramento do processo de adaptação, manutenção e evolução de sistemas customizados. Nesse estudo, foram conduzidos experimentos com amostras de variações reais com o objetivo de avaliar técnicas de LPS. Nesse trabalho foi implementado um conjunto de variações do SIG, adotado pela UFS (PASSOS; NETO, 2013). Entre as técnicas avaliadas nesse trabalho, estavam as abordagens anotativas, transformacional, de modelagem e composicionais. Em abordagens composicionais, como Programação Orientada a Aspectos(POA), as funcionalidades são implementadas separadamente em módulos distintos.

A Programação Orientada a Aspectos (POA) é um paradigma de programação que possibilita a separação do código de acordo com a sua importância para a aplicação (*separation of concerns*). Alguns paradigmas de programação não possibilitam que certos requisitos, os quais são denominados requisitos ou interesses transversais (*crosscutting concerns*), sejam adequadamente modularizados. Essa limitação ocorre quando a implementação de funcionalidades tende a se espalhar pelo código, entrelaçando-se com outras, aumentando o acoplamento e reduzindo a coesão do software (KICZALES *et al.*, 2001). Com a proposta de modularizar e encapsular corretamente os requisitos transversais, por meio de Aspectos, Kiczales *et al.* (1997) apresenta a POA. Na POA, os requisitos transversais são tratados como uma dimensão distinta dos demais do sistema e possibilita a modularização de interesses transversais.

A adoção de uma solução baseada em aspectos para melhoria do processo de manutenção evolução de sistemas integrados de gestão pode trazer uma série de vantagens sobre a abordagem adotada atualmente, dentre as quais foram observadas (PASSOS; NETO, 2012; PASSOS *et al.*, 2014):

1. A realização de customizações necessárias no sistema original mediante a composição de módulos com o código base do sistema, sem a introdução de alterações nele;

2. A separação entre o código base fornecido pela UFRN e as modificações modularizadas nos Aspectos facilita a visualização do código que foi modificado e pertence à UFS;

3. A implementação das variações de forma modular e facilitar a gestão das variações diante da modularização sem refatorações no código base.

Nesses trabalhos foram obtidos resultados que indicam que AspectJ tornou o processo de atualização da versão do sistema customizado mais ágil, a partir do momento em que as variações ficam modularizadas nos Aspectos.

Apesar disso, as conclusões do trabalho são limitadas por terem sido feitas sobre uma amostra proporcional aos tipos de variações encontradas em módulos do SIG.

Diante disso, demonstra-se a importância realizar um estudo de caso com a finalidade de avaliar a adoção da Programação Orientada a Aspectos no processo de adaptação, manutenção e evolução em sistemas customizados da Universidade Federal de Sergipe. Como contribuição resultantes do desenvolvimento desse trabalho destaca-se a condução do estudo de caso em um cenário real, em um ambiente largamente utilizado, cuja dificuldade para manutenção e customização afeta várias instituições da administração pública federal adquirentes do sistema. Além disso, destaca-se a possibilidade de avaliar os benefícios da adoção da POA neste cenário, oferecendo a essas instituições material comparativo para avaliar a adoção dessa técnica.

Nas próximas seções deste capítulo, serão apresentadas a justificativa para realização do trabalho, os objetivos gerais e específicos, as contribuições esperadas e a organização da dissertação.

1.1 Justificativa

O gerenciamento de variações em sistemas customizados é uma atividade crítica, cuja realização torna-se ainda mais difícil por conta da evolução dos sistemas originais. As abordagens existentes precisam tratar tanto da questão evolutiva quanto da frequência e do espalhamento dessas modificações, que acabam afetando diversos pontos de código do sistema base.

Desse modo, é evidente a necessidade de abordagens para facilitar o processo de adaptação e manutenção, cuja eficiência e eficácia sejam cientificamente comprovadas em ambientes reais ou que simulem condições reais. Assim, evidencia-se a necessidade de avaliar e aprimorar técnicas e abordagens que possibilitem:

- a implementação de variações de forma localizada e sem modificações no código do sistema original;
- o desenvolvimento independente e paralelo de variações no sistema customizado;
- a redução da problemática das atualizações do sistema customizado mediante novas versões do sistema original;
- o auxílio aos desenvolvedores no gerenciamento de variações e na atualização de versões de um sistema customizado.

Diante deste cenário, justifica-se a realização deste estudo de caso com a finalidade de avaliar a adoção da Programação Orientada a Aspectos em um contexto real, verificando se a mesma proporciona uma melhoria significativa no processo de adaptação e manutenção em sistemas customizados adotado atualmente pela Universidade Federal de Sergipe.

1.2 Objetivos

O objetivo geral deste trabalho consiste em realizar um estudo de caso com a finalidade de avaliar o impacto da utilização da Programação Orientada a Aspectos como abordagem para implementação de variações na adaptação, manutenção e evolução de sistemas no contexto dos Sistemas Integrados de Gestão da Universidade Federal de Sergipe.

1.2.1 Objetivos Específicos

Além do objetivo principal, foram estabelecidos os seguintes objetivos específicos:

- Analisar, identificar e selecionar variações introduzidas pela UFS em um módulo do SIG.
- Utilizar a Programação Orientada a Aspectos para a implementação das variações selecionadas e aplicá-la no processo de adaptação e manutenção de um módulo do SIG.
- Comparar o uso da Programação Orientada a Aspectos para a implementação das variações selecionadas em comparação ao processo atual de adaptação e manutenção adotado pela UFS.
- Avaliar os resultados obtidos com a aplicação da POA para implementação das variações selecionadas.
- Realizar a atualização de versão do módulo selecionado, permitindo a evolução do mesmo, utilizando-se as variações desenvolvidas previamente em AspectJ para introduzir as customizações produzidas pela UFS em uma versão atual do módulo desenvolvida pela UFRN.
- Avaliar os resultados obtidos a partir da realização do estudo de caso, verificando se a adoção da POA na adaptação, manutenção e evolução de sistemas customizados em comparação ao processo adotado atualmente pela UFS.
- Avaliar o impacto da adoção da POA sob o ponto de vista dos desenvolvedores da UFS.

1.3 Hipótese

Espera-se que a adoção da Programação Orientada a Aspectos possibilite um melhor gerenciamento das variações, por meio de uma implementação modularizada e que diminua a ocorrência de erros durante a atualização de versões do sistema.

1.4 Organização da Dissertação

Este trabalho foi organizado em 5 capítulos. Neste primeiro capítulo, foi apresentado o problema de pesquisa, a justificativa para sua realização, os objetivos do trabalho e a hipótese. Os demais capítulos estão divididos da seguinte forma:

- O Capítulo 2 apresenta a fundamentação teórica que embasa a realização deste trabalho, incluindo a Programação Orientada a Aspectos. São apresentados ainda os trabalhos que se relacionam com este estudo.
- No Capítulo 3, são apresentados os trabalhos que têm alguma relação com esta pesquisa.
- No Capítulo 4 é apresentada a metodologia adotada e os detalhes da realização do estudo de caso. Todos os aspectos do trabalho são definidos, tais como: como objetivo, planejamento, além das etapas e as questões de pesquisa que norteiam este trabalho.
- No Capítulo 5 são apresentadas as etapas e atividades realizadas para operação do estudo de caso.
- No Capítulo 6, a análise e a discussão dos resultados são apresentadas.
- Por fim, no capítulo 7, apresentam-se as conclusões obtidas a partir da realização deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo é apresentado um levantamento bibliográfico com os conceitos teóricos necessários para fundamentar a realização deste trabalho. Na primeira seção são apresentados os conceitos relacionados à Engenharia de Linhas de Produtos de Software. Em seguida a programação Orientada a Aspectos é discutida. Na seção seguinte, são apresentados detalhes dos Sistemas Integrados de Gestão, incluindo o processo de desenvolvimento e modularização de variabilidades adotado atualmente pela UFS.

2.1 Engenharia de Linhas de Produtos de Software

Uma Linha de Produtos de Software (*Software Product Line* - SPL) é um conjunto de sistemas de software com determinadas funcionalidades em comum, as quais satisfazem as necessidades de um segmento de mercado, e que são desenvolvidos a partir da mesma base. O conceito de linhas de produtos deriva de outras áreas da produção industrial em massa de bens de consumo, como automóveis e eletrônicos (POHL; BÖCKLE; LINDEN, 2005). O objetivo de uma Linha de Produtos de Software é minimizar o custo de desenvolvimento e evolução de produtos de software que façam parte de uma família (GURP; BOSCH; SVAHNBERG, 2001).

De acordo com Pohl, Böckle e Linden (2005), a engenharia de Linha de Produtos de Software (LPS) é um paradigma para o desenvolvimento de software por empresas a partir

de plataformas e customização em massa, tratando seus produtos como membros de uma mesma família que possuem várias funcionalidades em comum, as quais podem ser compartilhadas pelos sistemas de software que fazem parte de uma mesma LPS.

Desse modo, os desenvolvedores podem focar as características variáveis, isto é, a variabilidade, o que deve ser alterado ou customizado em cada sistema que faz parte da família (POHL; BÖCKLE; LINDEN, 2005), (GURP; BOSCH; SVAHNBERG, 2001). A variabilidade define o que pode ser customizado dentro da LPS, permitindo que sejam gerados diferentes produtos. Um gerenciamento adequado de variabilidades é necessário para garantir a integridade das aplicações que fazem parte da LPS e balancear suas características em comum e variáveis.

A variabilidade é responsável pelos pontos de variação, que são locais do artefato de software onde uma variação do produto pode acontecer (JACOBSON; GRISS; JONSSON, 1997; GURP; BOSCH; SVAHNBERG, 2001). Jacobson, Griss e Jonsson (1997) define ponto de variação como "um ou mais locais em que a variação vai ocorrer". Variantes são alternativas associadas ao ponto de variação. A configuração do produto permite escolher os pontos de variação e as variantes que irão compor um produto de software (POHL; BÖCKLE; LINDEN, 2005).

O gerenciamento de variabilidade possibilita a existência dos diferentes produtos em uma LPS. De acordo com (DURSCKI *et al.*, 2004), a adoção de linhas de produtos permite que uma organização obtenha diversos benefícios, como lançar um produto em menos tempo no mercado, com uma qualidade maior, e ainda aumentar a sua produtividade. O tempo para lançar novos produtos é diminuído pois a introdução de variabilidade em um já existente é feita de forma mais rápida que construir um sistema inteiro. O aumento de qualidade se deve à capacidade de customização em massa, diminuição dos erros e alta detecção de defeitos, pois os artefatos são constantemente adaptados, testados e verificados dentro da LPS. O aumento da produtividade se deve a uma reutilização eficiente dos artefatos de software.

No trabalho de Alves (2007), é apresentado um método para criação e evolução de LPS, utilizando a Programação Orientada a Aspectos. Na próxima seção esse paradigma de programação é discutido em detalhes.

2.2 Programação Orientada a Aspectos

Algumas funcionalidades não respeitam de forma adequada a modularização fornecida pela Programação Orientada a Objetos (POO). Esses interesses ou funcionalidades são denominados requisitos transversais (*crosscutting concerns*) (KICZALES *et al.*, 1997) e podem se espalhar ou entrelaçar por todo o código (ELRAD; FILMAN; BADER, 2001). Sem o uso de técnicas apropriadas para separar esses interesses e modularizá-los corretamente, pode-se observar a ocorrência de código entrelaçado (*code tangling*), quando múltiplos interesses são executados simultaneamente e de código espalhado (*code scattering*), quando um interesse é implementado em vários módulos (LADDAD, 2003). A ocorrência de código entrelaçado e espalhado afeta o desenvolvimento, resultando em forte acoplamento entre classes, fraca coesão, redundância e prejuízos na compreensão, manutenção e reúso do software (ELRAD; FILMAN; BADER, 2001).

Com a proposta de modularizar e encapsular corretamente esse tipo de requisito, Kiczales *et al.* (1997) apresenta a Programação Orientada a Aspetos (POA). Na POA são utilizados Aspectos como uma forma de modularizar os interesses transversais, tratando-os como uma dimensão distinta dos demais requisitos do sistema.

De acordo com Kiczales *et al.* (2001), a POA fez pelos interesses transversais o mesmo que a POO fez por um objeto, com o encapsulamento e a herança, fornecendo mecanismos de linguagem para capturar explicitamente requisitos transversais. Através da POA, interesses transversais podem ser escritos de forma modular, possibilitando a escrita de código mais simples e mais fácil de desenvolver e manter, ainda com maior potencial de reutilização.

Os principais elementos da POA são:

- *Aspect*: Unidade de programação, captura os requisitos transversais e a funcionalidade que entrecorta a aplicação.
- *Joinpoint*: Define os pontos onde os aspectos irão interceptar a execução do programa.
- *Pointcut*: Responsável por capturar os *joinpoints* de acordo com um conjunto de critérios.

- *Advice*: Trechos de código executados quando uma interceptação é realizada.
- *Inter-type*: declarações que permitem ao desenvolvedor adicionar campos, métodos, ou alterar o relacionamento entre classes, interfaces e classes abstratas. Esse tipo de declaração opera estaticamente, em tempo de compilação.

Um *advice* pode assumir os seguintes comportamentos:

- *before*: executa antes do *joinpoint*;
- *after*: executa depois do *joinpoint*;
- *around*: pode substituir o código interceptado pelo *joinpoint*.

Uma proposta para trabalhar com a POA é a extensão orientada a aspectos da linguagem Java: *AspectJ* [1]. Esta oferece suporte para a implementação modular de interesses transversais (KICZALES *et al.*, 2001). Os **Aspectos** constituem a unidade básica da linguagem e são definidos por declarações similares às de **Classes** em Java.

AspectJ suporta dois tipos de implementações transversais: dinâmica e estática. A primeira permite que sejam definidas execuções adicionais em pontos que podem ser interceptados durante a execução do programa. A implementação estática permite definir novas operações sobre tipos existentes, através de especificações de classes e comportamentos que podem ser utilizados pelos interesses dinâmicos (LADDAD, 2003). *Pointcuts* e *advices* afetam dinamicamente o fluxo do programa, as declarações *inter-type* afetam estaticamente a hierarquia de classes de um programa, e os aspectos encapsulam essas novas construções (LADDAD, 2003).

Através da implementação de um editor de figuras, é possível exemplificar como é possível utilizar a Programação Orientada a Aspectos com AspectJ (ELRAD; FILMAN; BADER, 2001). Na Figura 2.1 é apresentado o diagrama UML do editor de figuras.

Um elemento `Figura` é composto por uma série de `ElementoDeFigura`, que podem ser do tipo `Ponto` ou `Linha`. Abaixo é apresentada a implementação em Java das classes `Linha` na listagem 2.1 e `Ponto` na listagem 2.2, sem a utilização da POA.

[1] `http://www.eclipse.org/aspectj/doc/released/progguide/starting.html`

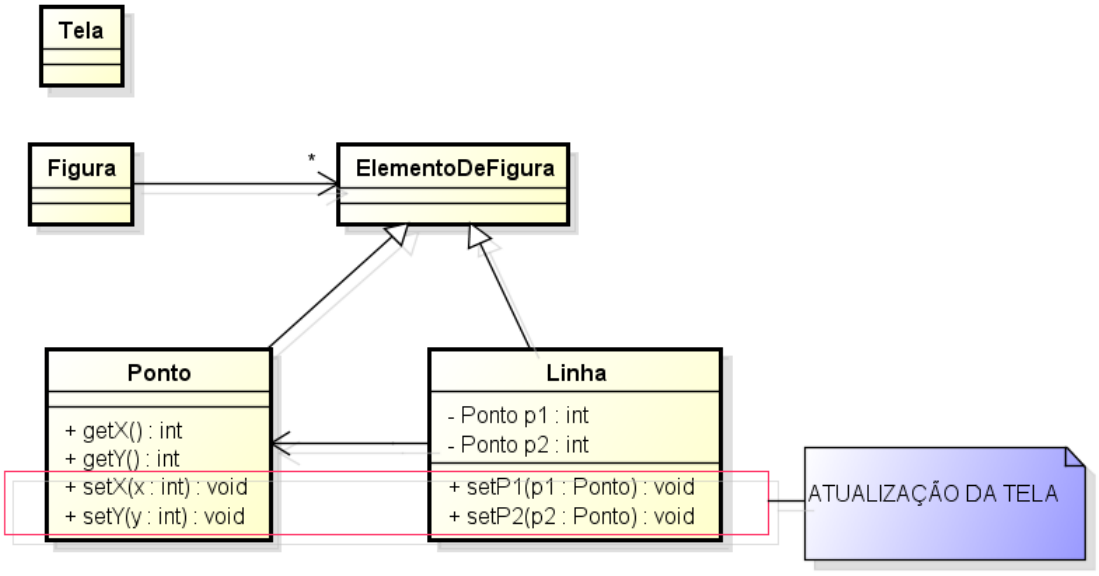


Figura 2.1: Diagrama UML do Editor de Figuras (ELRAD; FILMAN; BADER, 2001)

```
1  public class Linha{
2    private Ponto p1;
3    private Ponto p2;
4    public void setP1 (Ponto p1){
5      this.p1 = p1;
6      Tela.atualiza();
7    }
8    public void setP2 (Ponto p2){
9      this.p2 = p2;
10     Tela.atualiza();
11   }
12 ...
13 }
```

Listagem 2.1: Implementação da classe `Linha`

```
1   public class Ponto{
2     private int x = 0;
```

```
3       private int y = 0;
4       public void setX(){
5         this.x = x;
6         Tela.atualiza();
7       }
8       public void setY(){
9         this.y = y;
10        Tela.atualiza();
11      }
12      ...
13    }
```

Listagem 2.2: Implementação da classe `Ponto`

É possível observar que, após a chamada dos métodos `setX` e `setY` da classe `Ponto` e dos métodos `setP1` e `setP2` da classe `Linha`, é necessário realizar uma atualização da tela, através da invocação do método `atualiza` da classe `Tela`. Ou seja, sempre que os elementos da figura forem movimentados, será necessário atualizar a tela. A chamada ao método `atualiza` da classe `Tela` encontra-se **espalhada** pelos métodos da classe `Linha` e `Ponto`, por isso pode ser considerada um interesse transversal.

É possível, utilizando a Programação Orientada a Aspectos, realizar a modularização adequada desse interesse, encapsulando-o em um `Aspecto`. Para isso, é necessário:

- identificar os *joinpoints*: nesse caso, a invocação dos métodos que movem figuras do editor;
- especificar os *pointcuts* por meio de conjuntos de *joinpoints*;
- especificar o código que será executado nos *joinpoints* definidos.

Abaixo, na listagem 2.3, é apresentada a implementação em *AspectJ* de um aspecto `AtualizaTela` que modulariza o interesse transversal desse exemplo. Na listagem são apresentadas as classes •ttPonto e `Linha` do editor de figuras, em que o interesse transversal foi removido do código e o aspecto foi modularizado `AtualizaTela`.

```
1   aspect AtualizaTela{
2     pointcut mover():
3         call(void Linha.setP1(Ponto)) ||
4         call(void Linha.setP2(Ponto)) ||
5         call(void Ponto.setX(int)) ||
6         call(void Ponto.setY(int));
7       after() returning: mover(){
8         Tela.atualiza();
9       }
10  }
```

Listagem 2.3: Implementação do Aspecto `AtualizaTela`

Após o encapsulamento da linha de código que implementava o interesse transversal de atualizar a tela, no aspecto da listagem 2.3, é possível atualizar as classes Ponto e Linha, removendo essa chamada `Tela.atualiza();` dos métodos setP1, setP2, setX e setY. Com isso, promove-se a separação do interesse transversal e evita-se o espalhamento e entrelaçamento do código, modularizando-se essa funcionalidade.

Considerando-se a necessidade de inserir um requisito compartilhado a algumas classes existentes que já fazem parte de uma hierarquia de classes, isto é, que já estende outra classe. Em Java, é possível criar uma interface que captura esse novo requisito e adicionar a cada classe afetada um método que implemente essa interface. AspectJ permite expressar esses requisitos em apenas um lugar com declarações *inter-type* (ECLIPSE, 2015). Em um aspecto são declarados os métodos e campos necessários para implementar o requisito e esses são associados às classes existentes. Supondo que haja a necessidade de ter objetos do tipo `Tela` observando mudanças de objetos do tipo `Ponto`, e `Ponto` é uma classe já existente. É possível implementar esse requisito escrevendo um aspecto que define que a classe Ponto tem um campo de instância do tipo Ponto, os `observadores`, que mantém o controle dos objetos de Tela que estão observando os `Pontos`. Na listagem 2.4 é apresentado um exemplo dessa implementação, utilizando declarações *inter-type*.

```
1  aspect ObservadorDePonto {
2      private Vector Ponto.observadores = new Vector();
3      public static void adicionarObservador(Ponto p, Tela t) {
4          p.observadores.add(s);
5      }
6      public static void removerObservador(Ponto p, Tela t) {
7          p.observadores.remove(s);
8      }
9      pointcut altera(Ponto p): target(p) && call(void Ponto.set*(int
           ));
10     after(Ponto p): altera(p) {
11         Iterator iter = p.observadores.iterator();
12         while ( iter.hasNext() ) {
13             atualizaObservador(p, (Tela)iter.next());
14         }
15     }
16     static void atualizaObservador(Ponto p, Tela t) {
17         t.atualiza(p);
18     }
19 }
```

Listagem 2.4: Exemplo de declaração *inter-type* com AspectJ

O campo `observadores` é privado. Portanto, apenas o aspecto `ObservadorDePonto` pode vê-lo e os observadores são adicionados ou removidos nos métodos `adicionar Observador` e `removerObservador`. Um *pointcut* que define o que será observado (`altera`) e um *advice* que define o que será feito quando for observada uma mudança foram incluídos. Nota-se que não não foi necessário modificar as classes `Tela` ou `Ponto` e que todas as alterações necessárias para implementar este novo requisito são locais ao aspecto.

A Programação Orientada a Aspectos vem sendo aplicada em diversas áreas do desenvolvimento de software, como, por exemplo, *logging*, autenticação, validações (KICZALES *et al.*, 2001), e também na implementação de variabilidades em Linhas de Produtos de Soft-

ware (BRAGA *et al.*, 2007; ALVES *et al.*, 2006; PASSOS; NETO, 2013).

2.3 Sistemas Integrados de Gestão

Sistemas Integrados de Gestão (SIG) são sistemas de informação que integram dados e processos de uma organização em um único sistema (LAUDON; TRAVER, 2011). A Universidade Federal do Rio Grande do Norte (UFRN) vem desenvolvendo um conjunto de sistemas integrados de gestão da informação que tem por objetivo a integração e informatização da administração universitária e da administração pública federal (SINFO/UFRN, 2013). Esses sistemas estão sendo adotados por diversos órgãos da Administração Pública Federal (APF), incluindo órgãos do ciclo de gestão, universidades e institutos federais de educação. Esses sistemas foram desenvolvidos e implantados primeiramente na UFRN e foram adquiridos por outros órgãos da administração pública federal. Porém, cada um desses órgãos precisa adequá-los de acordo com as suas necessidades, realidade e contexto.

Na Figura 2.2 é apresentada uma rede de cooperação técnica estabelecida entre instituições federais de ensino superior e órgãos de APF – como Ministério da Justiça, Polícia Federal e Rodoviária Federal, Agência Brasileira de Inteligência e Controladoria Geral da União — para promover a colaboração e interoperabilidade entre esses órgãos, com a da transferência de tecnologia da UFRN e capacitação para implantação e customização do SIG pelos órgãos públicos envolvidos, com a finalidade de adaptar e customizar o SIG para sua realidade e contexto [2].

O conjunto de sistemas fornecidos pode ser agrupado em: sistemas para gerenciamento da área fim da universidade, como o sistema acadêmico, a sistemas da área meio, que tem como finalidade possibilitar uma administração eficiente dos recursos, incluindo a gestão de pessoas, patrimônio, planejamento, entre outros. Esses sistemas vem sendo desenvolvidos de forma integrada, permitindo uma comunicação eficiente entre os setores administrativos da instituição e também com outros sistemas da administração pública federal.

Na Figura 2.3 é apresentado um diagrama de inter-relacionamento entre o SIG, os siste-

[2]Imagem extraída de `http://www.info.ufrn.br/html/conteudo/cooperacoes/`

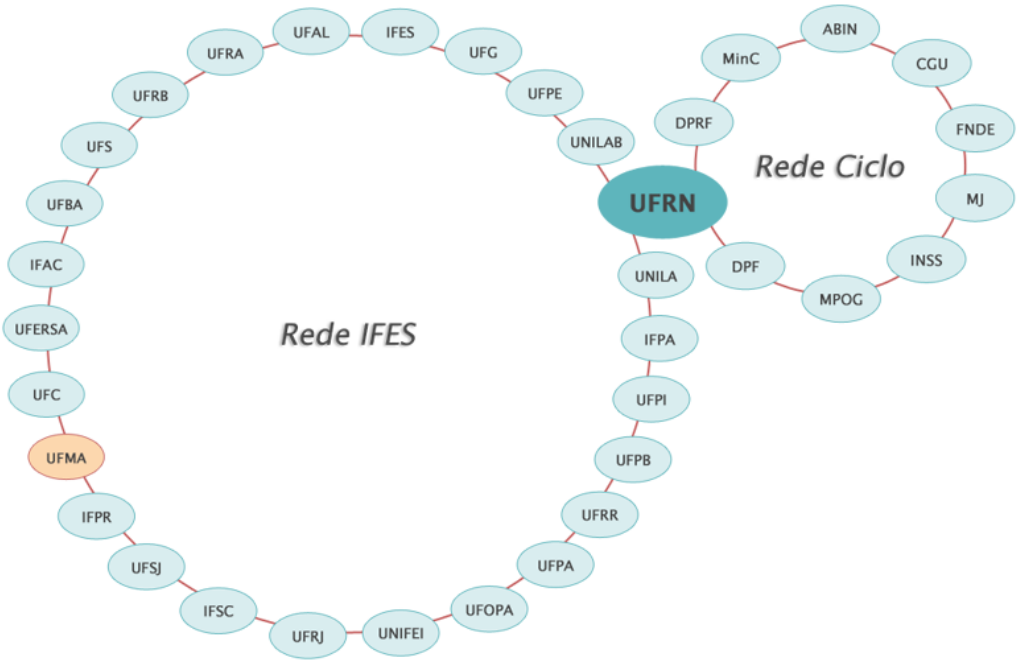


Figura 2.2: Cooperação técnica entre instituições da APF para implantação e adaptação do SIG.

mas estruturantes governamentais e as suas funcionalidades [3].

Os sistemas desenvolvidos pela SINFO/UFRN, compreendem os seguintes subsistemas, módulos e funcionalidades (SINFO/UFRN, 2013):

- Sistema Integrado de Gestão de Atividades Acadêmicas (SIGAA).
- Sistema Integrado de Patrimônio, Gestão e Contratos (SIPAC).
- Sistema Integrado de Gestão e Recursos Humanos (SIGRH).
- Sistema Integrado de Gestão de Planejamento e de Projetos (SIGPP).
- Sistema Integrado de Gestão Eletrônica de Documentos (SIGED).
- Sistema Integrado de Gestão da Administração e Comunicação (SIGAdmin).

[3] Imagem extraída de `http://www.info.ufrn.br/html/conteudo/cooperacoes/`

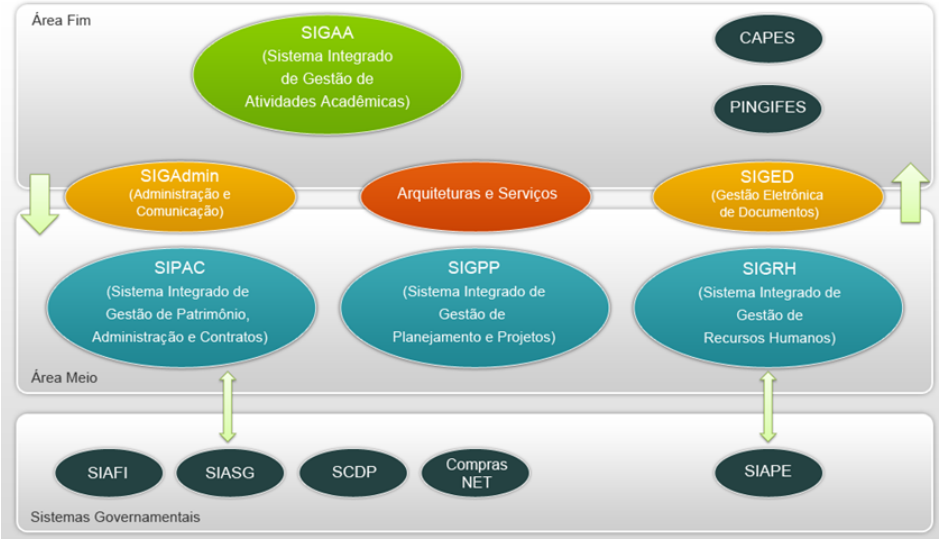


Figura 2.3: Diagrama de inter-relacionamento do SIG.

2.3.1 Desenvolvimento do SIG

O SIG foi desenvolvido usando o paradigma da Orientação a Objetos e Padrões de Projeto (GAMMA *et al.*, 1995). Os sistemas foram desenvolvidos em Java, tendo por base uma arquitetura em comum que permite abstrair a complexidade da implementação de sistemas JEE como também definir padrões de codificação, visualização e navegação para os sistemas WEB (SINFO/UFRN, 2013; ALUR; CRUPI; MALKS, 2003).

2.3.2 Modularização de variabilidades no SIG

A necessidade de adaptação e customização do SIG para a realidade da UFS incluindo novos requisitos, características e regras de negócio específicas levou a equipe de desenvolvimento a adotar um processo para introdução de variabilidades.

O processo atual adotado pela equipe de desenvolvimento da UFS consiste em partir do código fonte original e fazer alterações controladas no código fonte, demarcando os trechos que forem removidos, alterados e incluídos (PASSOS; NETO, 2012). O processo de alterações é administrado, utilizando-se a ferramenta de gerenciamento de projetos, o *Redmine* [4], na qual são controladas as solicitações por meio da criação *tickets* que possuem um código

[4] `http://www.redmine.org/`

de identificação único. Cada mudança realizada no código fonte original pela equipe de desenvolvimento da UFS é identificada pela inclusão no código da *tag* ***MUDANCA*** seguida do número do *ticket* e de uma descrição da alteração. Um exemplo de modificação introduzida pode ser visto na linha 6 da listagem 2.5. Essa estratégia adotada pela equipe aproxima-se de abordagens anotativas no contexto de uma LPS (APEL; KÄSTNER, 2009).

```
1  ...
2  private void popularPesquisa(PlanoTrabalho planoTrabalho) {
3    this.descricao = planoTrabalho.getTitulo();
4    this.id = planoTrabalho.getId();
5    this.responsavelProjeto = planoTrabalho.getOrientador();
6    //MUDANCA Ticket #11877
7    // Permitir mostrar interesse em projetos de docente externos
8    this.responsavelExternoProjeto = planoTrabalho.getExterno();
9    this.areaConhecimento = planoTrabalho.getProjetoPesquisa()
10                 .getAreaConhecimentoCnpq();
11 ...
```

Listagem 2.5: Exemplo de utilização da *tag* MUDANCA

Na figura 2.4 o processo de customização adotado pela UFS é exemplificado graficamente para uma melhor compreensão. A equipe da UFS realiza a clonagem do código original da UFRN e realiza as modificações diretamente no código fonte utilizando a *tag* **MUDANCA**. Estas modificações, realizadas independentemente das modificações feitas pela UFRN, ficam espalhadas e entrelaçadas com o código fonte original.

Este processo adotado pela referida equipe para customizar o sistema não vem se mostrando adequado, 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. Entre as desvantagens da utilização desta abordagem destaca-se a falta de delimitação precisa nas marcações, o controle das marcações recair completamente sobre os desenvolvedores, deixando o processo sujeito a falhas, a dificuldade em identificar e acompanhar as modificações introduzidas pela equipe e a dificuldade em realizar a atualização para novas versões dos sistemas (PASSOS; NETO, 2012). Na figura 2.5 é exemplificado

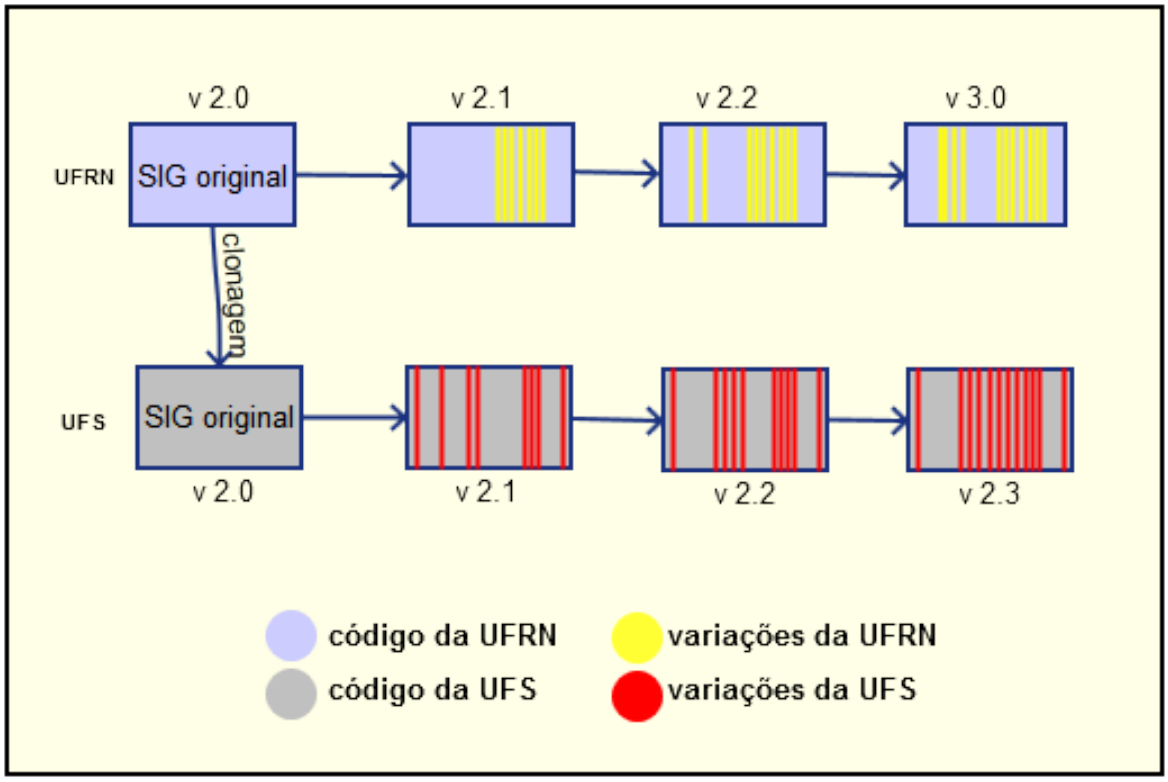


Figura 2.4: Processo de customização do SIG, variações entrelaçadas e espalhadas pelo código original

o processo de atualização para uma nova versão do sistema. É necessário realizar um processo de integração (*merge*) das versões, ao atualizar para uma nova versão do SIG original, inserindo as customizações realizadas pela equipe da UFS na nova versão do sistema atualizado. Contudo, as modificações realizadas pela UFRN na versão original e as customizações inseridas pela UFS podem entrar em conflito. Com isso, a análise que deve ser realizada para reintrodução das modificações torna-se bastante complexa e completamente dependente dos desenvolvedores. Esses devem realizar uma análise detalhada das variações para verificar a existência de conflitos.

Atualmente, a atualização para novas versões foi suspensa devido às dificuldades relatadas, principalmente pela necessidade de realizar manualmente o *merge* das versões, ou seja, reintroduzir todas as adaptações no código fonte de uma nova versão. Esse processo além de complicado, custoso e demorado torna-se ainda mais difícil pela possibilidade de exclusão de classes, trechos de código e modificações que impossibilitem a reintrodução dos trechos de código produzidos pela UFS.

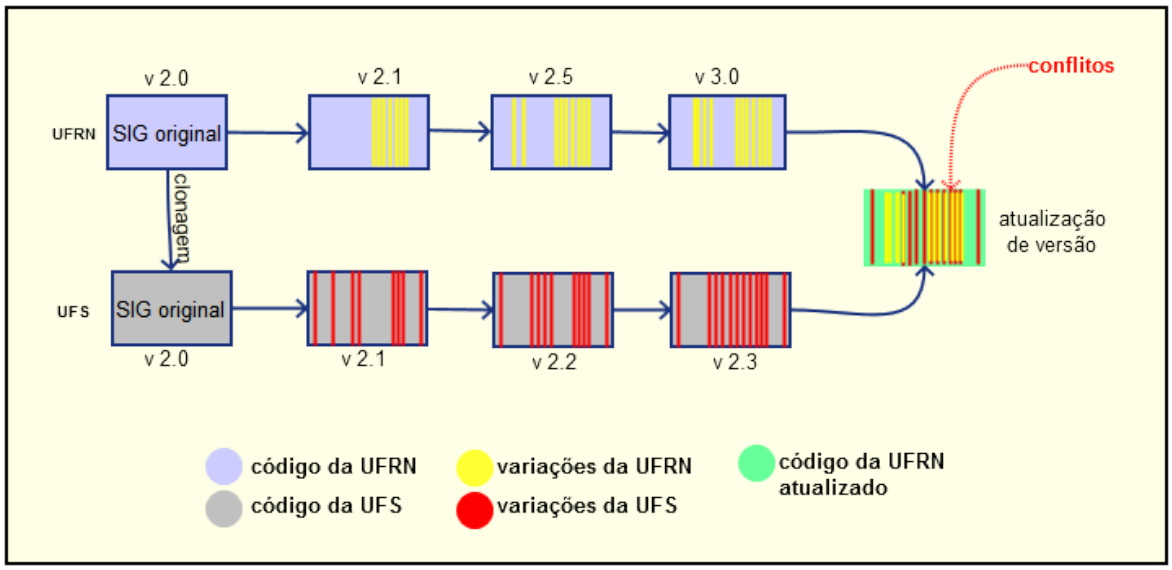


Figura 2.5: Processo de atualização de versão do SIG com conflitos de alterações

A necessidade de aprimorar esse processo de adaptação do SIG tem levado ao estudo de técnicas de modularização de variabilidades no contexto de linhas de produto de software para aplicação nesse problema (PASSOS *et al.*, 2014; SENA *et al.*, 2012; SANTOS *et al.*, 2012; LIMA *et al.*, 2013). Na próxima seção, o estudo de técnicas para implementação de variações no SIG conduzido por Passos *et al.* (2014) é detalhado.

2.3.3 Estudo de abordagens para implementação de variações em uma LPS

Um estudo sistemático sobre técnicas para lidar com variações em LPS é apresentado no trabalho de (PASSOS *et al.*, 2014). Os critérios definidos para selecionar as técnicas levantadas levaram em conta técnicas que permitissem o suporte à fase de implementação do ciclo de vida do software, à derivação de produtos, à criação de artefatos de software customizáveis com suporte ao nível de granularidade grossa e fina e à separação das variações do código-base do sistema original. Essas abordagens foram classificadas entre as categorias anotativa, composicional, modelagem e transformacional.

- As abordagens **anotativas** utilizam anotações em artefatos, modelos ou em uma base de código comum para configurar variantes estaticamente antes da compilação (APEL;

KÄSTNER, 2009).

- A abordagem **composicional** permite a implementação de funcionalidades em módulos distintos (arquivos, classes, pacotes, *plug-ins*) e outros) separadamente e, posteriormente, a composição destes gera as instâncias da linha de produtos.

- Na abordagem de **modelagem** é possível especificar as funcionalidades que contemplam o escopo de LPS, incluindo as dependências, os relacionamentos e as restrições entre elas. Nessa abordagem, o processo de derivação de produtos que envolve a seleção, a composição e customização dos artefatos de código é sistematizado com o objetivo de atender a configuração de funcionalidades.

- Abordagem **transformacional**: nesta, os sistemas de transformação de programa possibilitam a realização de modificações arbitrárias como a transformação de um programa em um novo com a mesma linguagem-fonte ou em um novo com uma linguagem diferente.

Nesse trabalho, foram apresentadas técnicas que lidam com variações em LPS e que podem ser usadas por equipes diferentes e independentes de desenvolvimento na realização de customizações ou adaptações de um sistema sem refatoração do código-base. Entre as técnicas levantadas diante das abordagens estudadas, as composicionais AspectJ e FeatureHouse e anotativa XVCL foram avaliadas e comparadas em relação à abordagem atualmente adotada para customização do SIG, na Universidade Federal de Sergipe. O trabalho consistiu na execução de um experimento em duas etapas. Na primeira, foram implementadas as variações selecionadas nas quatro abordagens e avaliados o tempo gasto para efetuar as adaptações e o grau de gravidade de erros cometidos durante elas. Na segunda, foi analisado o tempo gasto na atualização de versão do SIG customizado e o grau de erros ocorridos na sincronização de versão do sistema customizado após evolução do SIG original. Entre os resultados obtidos com a realização desses experimentos, foi possível observar que a técnica AspectJ não apresentou um desempenho superior em relação ao tempo necessário para implementar as variações. Foi observado ainda que AspectJ apresentou um maior índice de erros quando comparada às outras técnicas para implementação das variações selecionadas. Contudo, esses resultados são justificáveis pelo maior nível de complexidade necessário

no processo, pois a implementação de variações com AspectJ exige uma maior análise do código-base para a criação dos Aspectos. A conclusão apresentada no trabalho é que os desenvolvedores precisam realizar uma avaliação maior e mais cuidadosa para implementar os *pointcuts* adequados e ao escolher os *join points* que deverão ser interceptados, além do contexto que deverá estar disponível no *advice* para implementação de variação. Outra conclusão, em relação à adoção do AspectJ, foi sobre o ganho qualitativo proporcionado pela técnica que permite a criação dos artefatos customizáveis e a modularização das variações, mantendo-se o código-base do sistema original intacto. Na figura 2.6 é apresentado como funcionaria o processo de customização utilizando a POA. Observa-se que as modificações introduzidas anteriormente direto no código fonte, agora ficariam modularizadas pelos Aspectos, enquanto o código original não seria modificado.

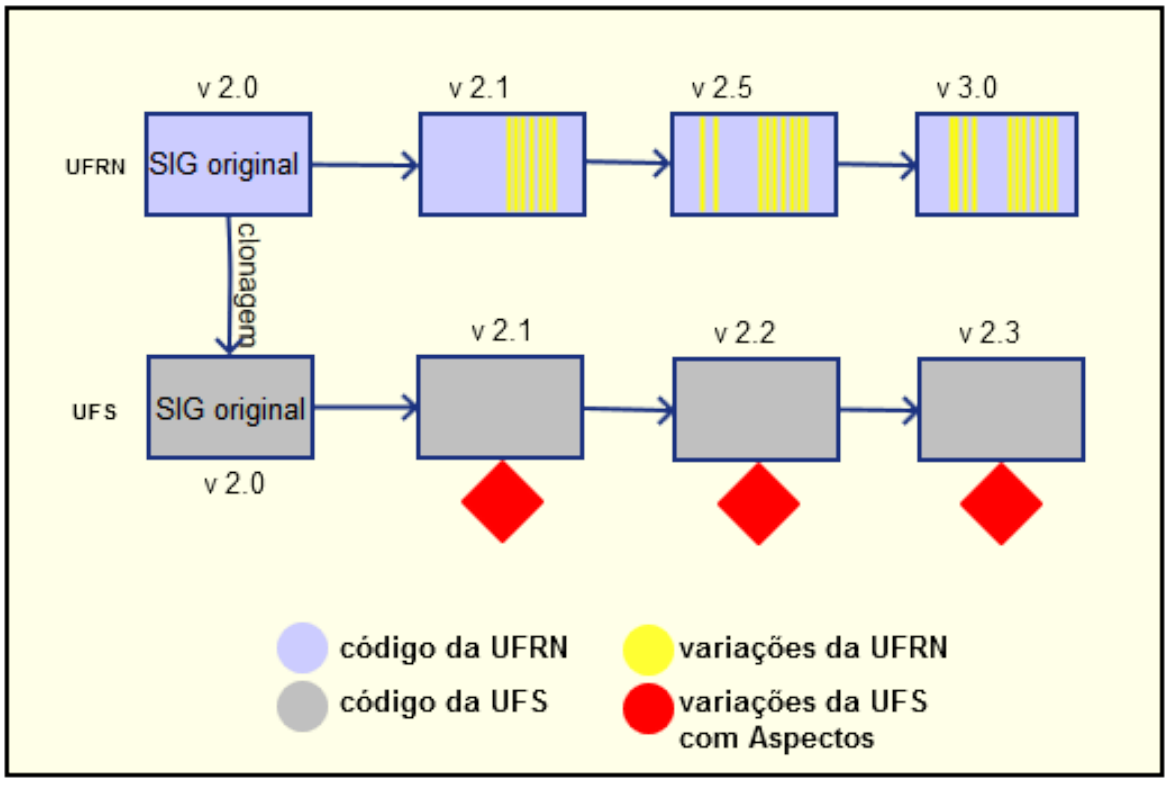


Figura 2.6: Processo de customização do SIG com Aspectos

Para exemplificar o processo de customização atual com a *tag* **MUDANCA** e o processo proposto com o uso de AspectJ, abaixo é apresentada a implementação da variação com o *ticket* 10031. Esse *ticket* envolve a solicitação de agrupar planos de trabalho por projeto na listagem de "‘Meus Planos de Trabalho"’. Na figura 2.7 é apresentado um ex-

certo da classe `PlanoTrabalhoDao.java` com a modificação realizada pela equipe da UFS. No lado esquerdo da figura podemos ver a *tag* **MUDANCA** com o número do *ticket* 10031, seguido da linha de código que modifica a ordenação dos planos de trabalho por projetos. No lado direito da Figura, encontra-se o código fonte original da classe intacto. É possível observar que a modificação é inserida diretamente sobre o código fonte original, como foi discutido na seção anterior. Na listagem 2.6 é apresentada a implementação da mudança utilizando-se AspectJ. A ordenação dos planos de trabalho é implementada no Aspecto `Variacao10031.aj`, de modo que o código fonte da classe `PlanoDeTrabalhoDao.java` permanece intacto.

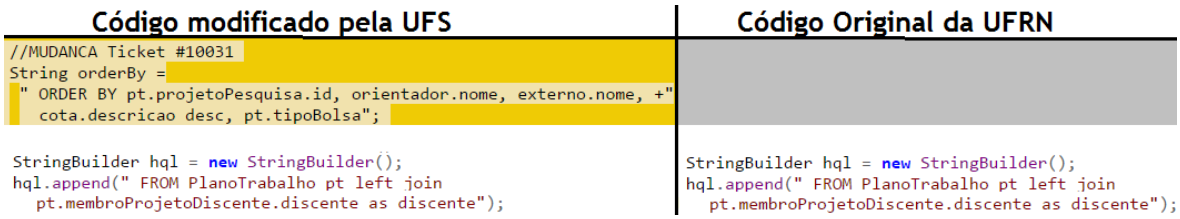


Figura 2.7: Implementação da variação 10031 na classe PlanoTrabalhoDao.java

```
1   public aspect Variacao10031 {
2   private String campoOrdenacao = " pt.projetoPesquisa.id, ";
3   pointcut ordenacaoPlanoPorProjetos(String sql):
4       call (public Query org.hibernate.Session.createQuery(
            String))
5        && args(sql) &&
6       withincode (public * br.ufrn.sigaa.arq.dao.pesquisa.
            PlanoTrabalhoDao.filter(..));
7   Object around(String sql) : ordenacaoPlanoPorProjetos(sql) {
8       StringBuilder sb = new StringBuilder(sql);
9       if (sql.indexOf("ORDER BY") >= 0){
10        sb.insert(sb.indexOf("ORDER BY") +
11        ("ORDER BY").length(),campoOrdenacao );
12      }
13      return proceed (sb.toString()) ;
14  }
15  }
```

Listagem 2.6: Implementação da alteração 10031 com Aspectos

Em relação ao processo de atualização do SIG customizado para uma nova versão do SIG original, a adoção de AspectJ apresentou um melhor desempenho quando comparada às outras técnicas avaliadas no experimento. Como mostrado na Figura 2.8 a utilização de Aspectos para modularizar as variações introduzidas pela UFS retira as customizações que estavam espalhadas e entrelaçadas com o código fonte original. Com isso, ao realizar uma atualização, é necessário apenas realizar a integração entre os Aspectos e a nova versão do código fonte original. A análise das customizações ficaria concentrada nos Aspectos, tornando o processo de atualização menos complexo do que ao utilizar a abordagem atual (PASSOS *et al.*, 2014). Deste modo, a conclusão obtida nesse experimento, evidencia a importância de avaliar a adoção da POA neste estudo de caso.

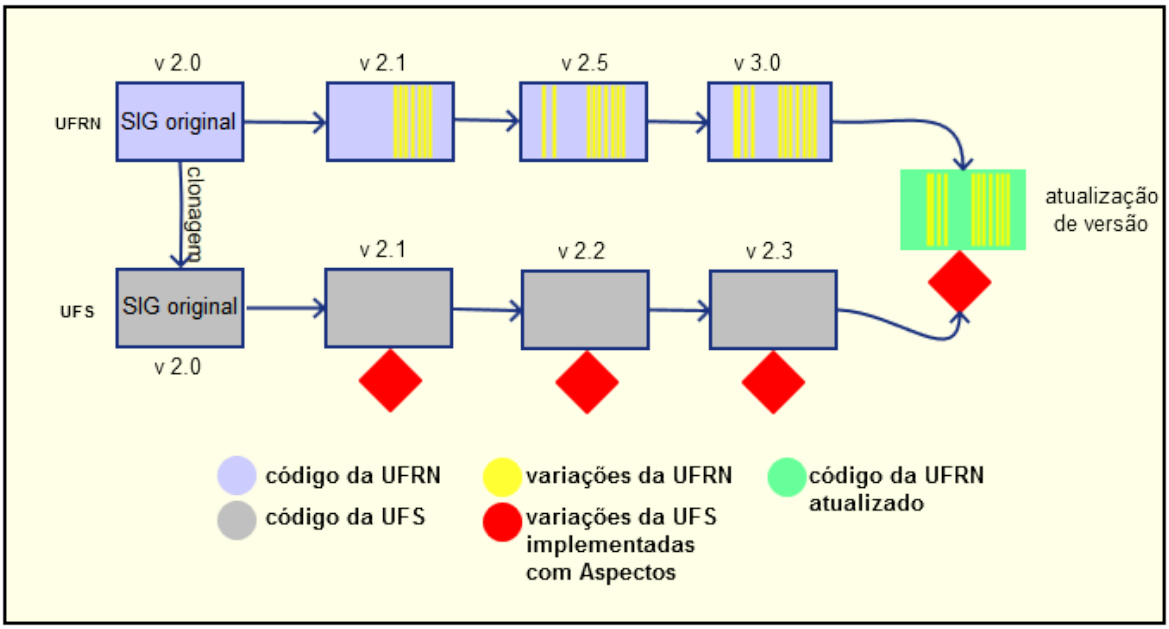


Figura 2.8: Atualização de versão do SIG com variações implementadas em Aspectos

Capítulo 3

Trabalhos Relacionados

Uma abordagem orientada a aspectos para o desenvolvimento de linhas de produtos de software é apresentada no estudo de caso conduzido por Braga *et al.* (2007). A abordagem proposta permite identificar os benefícios da adoção da POA para implementação de *features* em uma linha de produtos. Uma das conclusões obtidas nesse estudo é de que o encapsulamento das funcionalidades proporcionado pela POA possibilitam que estas se tornem mais coesas, facilitando a combinação e a reutilização. Contudo, nesse trabalho, é proposta uma abordagem de desenvolvimento incremental baseada em aspectos, que abrange desde o projeto inicial da linha de produtos, diferindo por isso, do que é proposto neste trabalho.

A utilização da POA para implementação de variações em LPS também é discutida por Alves *et al.* (2006). Nesse trabalho, é apresentado um estudo de caso para migração de uma LPS utilizando a POA. A POA é adotada para implementação das variabilidades como interesses transversais que ocorrem na LPS.

No trabalho de Sena *et al.* (2012) foi realizado um estudo sobre a adoção e composição de padrões de projeto tradicionais para implementar diferentes tipos de variabilidades no SIG. Esse estudo também avaliou cenários do uso de técnicas anotativas de execução condicional para implementar variabilidades de granularidade fina. A adoção de alguns padrões de projeto como *Strategy*, *Chain of Responsability* e *Template Method* (GAMMA *et al.*, 1995) foi avaliada na implementação de diferentes tipos de variabilidades. Uma das conclusões deste estudo foi que o uso de técnicas composicionais permitem a modularização

de variabilidades comportadas, ou seja, que ocorrem isoladas e são de granularidade grossa, podendo ser encapsuladas sem produzir efeitos colaterais nos outros artefatos da linha de produto e sem causar excessivo aumento de complexidade no código da mesma. Porém, outra conclusão refere-se a quando a variabilidade está espalhada em vários locais da LPS ou mesmo quando isolada em um único ponto, mas requer tratamento de granularidade mais fina, as técnicas anotativas surgem como uma boa opção. No trabalho de Lima *et al.* (2013) é proposta uma abordagem para a reintegração de linhas de produtos de software que evoluíram de forma independente a partir de uma versão inicial. A abordagem proposta permite a detecção automática de conflitos de LPS que evoluíram independentemente e a resolução e *merge* desses conflitos. Nesse trabalho apresenta-se um *plugin* desenvolvido para o Eclipse IDE que detecta automaticamente os conflitos nas LPS desenvolvidas independentemente e aponta qual a melhor estratégia a ser adotada para o *merge*: automática, semi-automática ou manual. Para avaliar essa proposta, foi realizado um estudo de caso com o código fonte do SIG customizado pela UFS e o código fonte original da UFRN. Um dos problemas evidenciados nesse trabalho refere-se justamente à necessidade de realizar uma atualização de versão após a introdução de modificações, problema que ocorre atualmente na UFS, uma vez que as modificações realizadas podem entrar em conflito com novas versões do código base original e fica a cargo do desenvolvedor resolver esses conflitos. Esse problema dificulta o processo de atualização e possibilita a introdução de erros e falhas no código. No caso do SIG, como atualmente cada instituição vem adotando processos de customização e adaptação próprios, a implementação de *merges* entre LPSs evoluídas de forma independente torna-se ainda mais complicada de ser realizada automaticamente.

Um estudo conduzido por Passos e Neto (2013) apresenta um levantamento e catalogação de variações introduzidas pela Universidade Federal de Sergipe (UFS) no código-base do SIG. Após a catalogação das variações, foram levantadas técnicas e abordagens adequadas para a introdução de variações no SIG. Técnicas para implementação de variações em LPS e que podem ser usadas por equipes de desenvolvimento na adaptação de um sistema sem alterações no código-base foram avaliadas em um experimento controlado (PASSOS *et al.*, 2014). As técnicas AspectJ, FeatureHouse e XVCL foram avaliadas e comparadas em relação à abordagem atualmente adotada para customização do SIG, na UFS. O trabalho consistiu na execução de um experimento em duas etapas. Na primeira etapa, foi realizada

a implementação de variações selecionadas nas quatro abordagens e avaliado o tempo gasto para efetuar as adaptações e o grau de gravidade de erros cometidos durante as implementações. Na segunda etapa do experimento, foi analisado o tempo gasto na atualização de versão do SIG customizado e o grau de erros ocorridos na sincronização de versão do sistema customizado após evolução do SIG original. Entre as técnicas estudadas, foram implementadas algumas variações codificadas em AspectJ. Observou-se que todas as variações encontradas e contabilizadas puderam ser convertidas em Aspectos com comportamento equivalente. Uma das conclusões apresentadas no trabalho é de que os desenvolvedores precisam realizar uma análise maior e mais cuidadosa para implementar uma variação utilizando AspectJ. A adoção de AspectJ apresentou um melhor desempenho quando comparada às outras técnicas avaliadas no experimento durante a atualização de versões Passos *et al.* (2014). Outra conclusão possível, diz respeito ao ganho qualitativo proporcionado pela adoção de AspectJ, pois essa técnica permite a criação dos artefatos customizáveis e a modularização das variações, possibilitando a alteração da estrutura estática e dinâmica do sistema, mantendo ainda o código-base original intacto. Neste estudo de caso, são considerados os resultados e conclusões obtidos a partir dos experimentos de Passos *et al.* (2014). Porém, como foi utilizada uma amostra proporcional aos tipos de variações encontradas em módulos do SIG, entende-se que os resultados da análise realizada nesse trabalho são limitados.

Foram encontrados também, os trabalhos de Rubin *et al.* (2012), Rubin, Czarnecki e Chechik (2013), apresentados a seguir. Nestes, discute-se o desenvolvimento de novos produtos para uma LPS de forma *ad-hoc*. Como, frequentemente, as empresas não desejam ou não podem arcar com uma refatoração de variantes de produtos clonados, Rubin *et al.* (2012) defendem que práticas de clonagem devem ser refinadas, considerando a questão sobre a reutilização efetiva de produtos como parte da engenharia da LPS, com base na clonagem de produtos (criação de novos produtos a partir de outros já existentes). Nesse trabalho, é utilizado um modelo de metadados (*Product Line Chageset Dependency Model*) que armazena informações sobre a linha de produtos desenvolvida em termos de funcionalidades, e permite a identificação de inconsistências na sua implementação, mantendo as LPS relacionadas por meio de um sistema de gestão de configuração de software para distinguir seções, permitindo tanto a reutilização de produtos existentes quanto a liberdade para modificá-los, auxiliando desenvolvedores ao clonarem ou excluírem funcionalidades de diferentes seções.

Em um trabalho posterior, Rubin, Czarnecki e Chechik (2013) conduziram um estudo em três empresas para fundamentar sua proposta de gestão de variantes de produtos clonados, ainda partindo do argumento de que, embora a engenharia de LPS esteja ganhando popularidade na indústria devido à sua promessa de aumentar o tempo de comercialização, reduzir custos e valorizar portfólio, entre outras coisas, muitas empresas continuam empregando clonagem na implementação de variabilidade de produtos. O estudo abordou a gestão tanto no processo de transição para uma estrutura de LPS quanto na manutenção de variantes de produtos clonados sem refatoração, defendendo que o sistema de operadores básicos de gestão poderia ser mais facilmente reutilizado e convidando a comunidade desenvolvedora de linhas de produtos a refiná-lo, aprimorando o *framework*. Por outro lado, apesar da demanda pela abordagem *clone-and-own* em empresas, uma outra forma de incentivar a reutilização de funcionalidades entre diferentes clones é tratá-las usando-se a LPS. Para tanto, foi desenvolvida uma ferramenta que analisa informações sobre a evolução da linha, extraindo-as dos sistemas de configurações e mudança (*issue tracking systems*), e procura organizá-las de forma a realizar um *merge* semiestruturado.

O estudo de Santos e Kulesza (2014), motivado pela aplicação bem-sucedida de LPS em diferentes empresas, pretende investigar os tipos e a quantidade de conflitos que surgem do *merge* do código-fonte de LPSs clonadas e evoluídas de forma independente, em relação ao tipo de técnica de modularização de variabilidades utilizada na LPS. Nesse trabalho explora-se a hipótese de que a clonagem e reconciliação de LPSs geram mais conflitos de código-fonte quando utilizam-se técnicas de modularização anotativas em vez de técnicas de modularização composicionais. Dessa forma, Santos e Kulesza (2014) pretendem identificar e refatorar variabilidades de uma linha de produtos usando duas técnicas de implementação de variabilidade e comparar os conflitos gerados usando uma ferramenta de *merge* que vem sendo desenvolvida pelo grupo de pesquisa. O objetivo do estudo é determinar quais técnicas de implementação de variabilidades geram menos conflitos ou conflitos mais fáceis de serem resolvidos ao se realizar o *merge* de linhas de produtos clonadas, incluindo o uso da técnica de *merge* semiestruturado, e aprimorar o estado atual do desenvolvimento da ferramenta de *merge* voltada para LPS, comparando os resultados obtidos pela análise de conflitos com os resultados de estudos prévios, melhorando sua precisão.

Um processo para avaliação quantitativa de refatorações em sistemas OO e OA, utilizando métricas de software é apresentado no trabalho de Pagliari e Nunes (2007). O processo definido tem como objetivo possibilitar uma avaliação quantitativa do impacto das refatorações sobre o código fonte alterado, utilizando métricas de análise quantitativa. A abordagem para análise do impacto de qualidade proposta nesse trabalho demonstra a importância da utilização de métricas quantitativas. Contudo, nesse trabalho são avaliadas refatorações de software, o que difere da proposta desta dissertação, uma vez que o código fonte original do SIG não deve ser modificado ou refatorado durante a implementação das variações.

A proposta deste trabalho se difere dos trabalhos mencionados por avaliar a utilização de AspectJ como uma técnica de LPS, por uma equipe de desenvolvimento responsável pela clonagem do código e introdução das modificações em apenas uma versão da LPS, mantendo o código-base original intacto.

Capítulo 4

Metodologia do Estudo de Caso

Esse trabalho foi desenvolvido por meio de uma perspectiva metodológica de natureza aplicada, uma vez que o conhecimento gerado foi aplicado a um fim específico. De acordo com Oliveira (1998), a pesquisa aplicada tem como objetivo "pesquisar, comprovar ou rejeitar hipóteses sugeridas pelos modelos teóricos e fazer a sua aplicação às diferentes necessidades humanas". Em relação ao objetivo, é possível classificar essa pesquisa como exploratória. Pesquisas exploratórias são aquelas em que se procura proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses, para aprimorar ideias ou descobrir intuições (GIL, 2002).

A pesquisa exploratória assume na maioria das vezes a forma de pesquisa bibliográfica ou de estudo de caso. Neste trabalho, é apresentado um estudo de caso (GIL, 2002). O estudo de caso é uma modalidade de pesquisa amplamente utilizada em pesquisas nas áreas médicas, psicológicas, humanas, tecnológicas e sociais (GIL, 2002; VENTURA, 2007). Um estudo de caso permite explorar o conhecimento de um fenômeno estudado a partir do estudo profundo e exaustivo de um ou poucos objetos, adquirindo-se amplo e detalhado conhecimento, o que seria impossibilitado pela adoção de outros delineamentos já considerados (GIL, 2002). Ventura (2007) afirma que um estudo de caso permite a investigação de um caso específico, bem delimitado, contextualizado e que é apropriado para pesquisadores individuais, pois possibilita a investigação do problema em profundidade com lugar e período de tempo limitado. Yin (2015) afirma que o estudo de caso é uma das estratégias favoritas

quando o foco da pesquisa se encontra em fenômenos contemporâneos inseridos em algum contexto da vida real. Essa estratégia de pesquisa, possibilita a observação do comportamento do objeto de estudo em um ambiente real e a coleta de dados qualitativos por meio de experiências, observações e entrevistas (EASTERBROOK *et al.*, 2008). Dentre as estratégias empíricas básicas para a organização e execução de estudos na engenharia de software experimental, Travassos e Gurov (2002) cita o estudo de caso, que pode ser utilizado para monitorar projetos, atividades e atribuições, observando-se um atributo específico. Runeson *et al.* (2012) define o estudo de caso na engenharia de software como:

> uma estratégia de pesquisa empírica que se baseia em múltiplas fontes de evidência para investigar um exemplo (ou um pequeno número de casos) de um fenômeno da engenharia de software dentro de um contexto de real, especialmente quando os limites entre o fenômeno e contexto não podem ser claramente especificados.

Runeson *et al.* (2012) estabelece um processo para condução do estudo de caso:

1. Planejamento do estudo de caso: quando os objetivos são definidos e a pesquisa é planejada;
2. Preparação para a coleta de dados: são definidos os procedimentos e protocolos para coleta de dados.
3. Coleta de dados: os procedimentos para a obtenção dos dados são executados.
4. Análise dos dados coletados: procedimentos de análise de dados são aplicados.
5. Elaboração de relatórios - o estudo e suas conclusões são empacotados em relatórios para apresentação.

4.1 Planejamento do Estudo de Caso

Segundo Ventura (2007), o planejamento do estudo de caso é geralmente organizado em torno de algumas questões relativas ao **como** e ao **porquê** da investigação.

Nesta seção, é apresentado o protocolo utilizado para a condução deste estudo de caso. Todas as informações necessárias para a realização e análise dos resultados desta pesquisa encontram-se definidos nesta seção.

4.1.1 Justificativa

A realização deste estudo de caso justifica-se pela necessidade de avaliar a adoção da Programação Orientada a Aspectos no contexto do processo de adaptação, manutenção e evolução de sistemas customizados, adotado atualmente pela Universidade Federal de Sergipe, verificando se a mesma proporciona uma melhoria significativa nesse processo.

4.1.2 Objetivos

O objetivo deste experimento consiste em avaliar o impacto da utilização da Programação Orientada a Aspectos como abordagem para implementação de variações na adaptação, manutenção e evolução de sistemas no contexto dos Sistemas Integrados de Gestão da Universidade Federal de Sergipe. O objetivo desse estudo de caso foi formalizado com base no modelo GQM (*Goal Question Metric*) (BASILI *et al.*, 2002): **Avaliar** a abordagem atual de desenvolvimento utilizada pela UFS contra a adoção da Programação Orientada a Aspectos para a adaptação, manutenção e evolução do Sistema Integrado de Gestão, **com respeito à** eficácia, do **ponto de vista de** desenvolvedores, **no contexto** da Universidade Federal de Sergipe (UFS).

4.1.3 Unidade de análise

Para a condução do estudo de caso foi selecionado o módulo de **Produção Intelectual** do **SIGAA**. Esse módulo, tem a finalidade de proporcionar a consulta de informações referentes à produção intelectual dos docentes. Foi realizada uma reunião com a equipe responsável pelo desenvolvimento do SIGAA, que faz parte da Coordenadoria de Sistemas (COSIS) do Núcleo de Tecnologia da Informação (NTI) da UFS. Durante essa reunião, foi apresentada a proposta deste trabalho e discutiu-se sobre qual módulo do SIGAA seria mais adequado

para a condução deste estudo de caso. Foi sugerido o módulo de Produção Intelectual do SIGAA, pois este já havia passado por intensas modificações no ano anterior, porém não estava sendo modificado atualmente pela UFS. Além disso, sabia-se que a UFRN também havia feito modificações recentes nesse módulo e que havia interesse por parte da UFS em incorporar estas modificações à versão atual modificada pela UFS. Assim, o módulo não estaria sendo modificado em paralelo pela UFS durante a realização deste estudo de caso e a atualização do módulo seria útil para a UFS posteriormente. Com isso, justifica-se a escolha do módulo de Produção Intelectual do SIGAA para realização do estudo de caso apresentado neste trabalho. Após a implementação do referido módulo utilizando AspectJ, prosseguiu-se com uma apresentação dos resultados aos desenvolvedores da COSIS para avaliação e discussão dos resultados.

4.1.4 Questões de pesquisa

Após a definição dos objetivos, foi possível elaborar as questões de pesquisa:

- O uso da Programação Orientada a Aspectos possibilitou a implementação de todas as variações introduzidas no módulo selecionado?
- Foi observada uma redução na ocorrência de erros na utilização dos Aspectos implementados durante o processo de atualização do SIG customizado para uma versão mais recente do SIG Original em relação a abordagem atual?
- Qual a opinião dos desenvolvedores a respeito da utilização de AspectJ em relação à abordagem atual?

4.1.5 Métricas

Para avaliação das questões de pesquisa, foram adotadas as seguintes métricas:

- Quantidade de variações catalogadas
- Quantidade de variações implementadas com AspectJ

- Erros detectados após atualização de versão
- Erros solucionados após a atualização de versão

4.1.6 Análise qualitativa

Para avaliação dos resultados da implementação com AspectJ, foi elaborado um questionário e definida a aplicação junto ao grupo de desenvolvedores que atualmente trabalham na manutenção do SIG na UFS.

O questionário, disponível no Apêndice , contém as perguntas apresentadas à equipe de desenvolvimento que trabalha atualmente com a manutenção do SIG. Esse questionário foi definido com o objetivo de caracterizar o nível de conhecimento da equipe em relação à linguagem e a técnica de desenvolvimento utilizada pela equipe atualmente, bem como de avaliar o nível de conhecimento em relação à abordagem proposta utilizando AspectJ e também qual a percepção da equipe no tocante à adoção de AspectJ dentro do processo de customização realizado atualmente pela UFS. Para isso, foram definidas 12 perguntas, entregues aos desenvolvedores após uma apresentação contendo os resultados e observações coletados nas etapas deste estudo de caso e alguns detalhes para compreensão de AspectJ. Os resultados da análise realizada com os dados coletados nesse questionário são apresentados na seção 6.4.

4.1.7 Instrumentação

Nesta seção são apresentadas as tecnologias e programas que serão utilizados para realização do estudo de caso, o que envolve o levantamento das variações e a configuração do ambiente de desenvolvimento.

- WinMerge [1]: é uma ferramenta código aberto para diferenciação e mesclagem no Windows. O WinMerge será usado para a comparação de arquivos facilitando a detecção e classificação das variações implementadas pela equipe de desenvolvimento da UFS no módulo selecionado para o estudo de caso.

[1] `http://winmerge.org/?lang=pt_br`

- Eclipse IDE for Java EE Developers [2]: é um ambiente integrado de desenvolvimento (Integrated Development Environment), que auxilia o desenvolvimento de aplicações Java para Web através do Java EE (*Java Enterprise Edition*), e foi adotado pela equipe de desenvolvimento original do SIG e pelas equipes de customização do SIG, incluindo a equipe da UFS. O Eclipse possui uma grande variedade de *plugins* que podem ser integrados à plataforma e oferecem suporte ao desenvolvimento de software.

- AspectJ Development Tools (AJDT) [3]: É uma ferramenta para o Eclipse IDE que oferece suporte ao desenvolvimento de Aspectos com AspectJ. O AJDT foi utilizado para apoiar a implementação dos Aspectos na realização do estudo de caso.

- PostgreSQL 9.3 [4]: É um Sistema Gerenciador de Banco de Dados Objeto Relacional (SGBDOR) livre e de código aberto. A versão 9.3 do PostgreSQL é utilizada pela equipe original de desenvolvimento do SIG para gerenciamento das bases de dados do sistema. Foram obtidos junto à equipe de desenvolvimento da UFS, os *scripts* necessários para criação da base de dados do SIGAA e coletada a base de dados atual do sistema para realização dos testes das implementações. Os dados obtidos foram previamente mascarados a fim de garantir a privacidade e segurança dos usuários do referido sistema.

- JBoss Aplication Server 4.2 [5]: É um servidor de aplicação de código fonte aberto baseado na plataforma JEE e implementado completamente na linguagem de programação Java, utilizado para realizar o *deployment* e gerenciamento de sistemas Web desenvolvidos com a plataforma JEE como o SIG. A versão 4.2 é adotada pela equipe de desenvolvimento do SIG e será utilizada na realização do Estudo de Caso.

4.2 Ameaças à validade

Para mitigar ameaças à validade do estudo, a unidade de análise escolhida para condução do estudo foi selecionada pela equipe de desenvolvimento da UFS, por ser considerada uma

[2] `https://eclipse.org/`
[3] `http://eclipse.org/ajdt/`
[4] `http://www.postgresql.org`
[5] `http://www.jboss.org/`

amostra representativa para análise do SIGAA. Todas as variações encontradas no módulo de Produção Intelectual foram catalogadas e selecionadas para implementação. Além disso, os testes das implementações foram realizados de forma automatizada, considerando o ambiente de desenvolvimento da UFS e todos os artefatos gerados durante a condução deste trabalho foram armazenados, possibilitando a reexecução do mesmo e reavaliação das hipóteses. Contudo, entende-se que devido às limitações da execução deste estudo de caso, as conclusões apresentadas neste trabalho não podem ser generalizadas para a introdução de variações em outros sistemas. Assim, o estudo de caso apresentado neste trabalho considera apenas o cenário do SIG, no contexto da Universidade Federal de Sergipe, que pode ser estendido à outras instituições que implantem o referido sistema e desejem customizá-lo.

Capítulo 5

Operação do Estudo de Caso

Neste capítulo, são apresentadas as atividades realizadas para condução do estudo de caso previsto neste trabalho. Na seção 5.1, é apresentada a unidade de análise do estudo em detalhes. Na seção 5.2, é apresentado o levantamento e análise das variações encontradas. Na seção 5.3, os detalhes da implementação das variações utilizando a POA são apresentados e na seção 5.4 detalhes da variação que não foi possível implementar com AspectJ. Na seção 5.5 são descritos os testes realizados para verificar as implementações. Por fim, na seção 5.6 é descrita a atualização do módulo utilizando a POA.

5.1 Módulo de Produção Intelectual do SIGAA

O módulo de Produção Intelectual permite cadastrar e gerenciar as atividades acadêmicas desenvolvidas pelos docentes, funcionando como uma espécie de currículo do docente. O módulo foi desenvolvido com a finalidade de proporcionar a consulta de informações referentes à produção intelectual dos docentes (SINFO/UFRN, 2013). É necessário que existam dados de docentes na base para que seja possível cadastrar as produções intelectuais. A produção intelectual do docente é utilizada como critério para a concessão de cotas de bolsas aos docentes vinculados aos "Projetos de Pesquisa".

5.1.1 Arquitetura do módulo

Na Figura 5.1, é apresentado um diagrama do módulo de Produção Intelectual que retrata os pacotes ou partes do sistema divididos em agrupamentos lógicos e as dependências entre eles.

A seguir, é apresentada uma descrição dos pacotes que compõem o módulo de Produção Intelectual e o diagrama mostrado na Figura 5.1.

- Pacote `br.ufrn.sigaa.arq.dao.prodocente`: Contém vinte e uma (21) classes DAOs (*Data Access Objects*) utilizadas pelas operações do módulo de Gerenciamento de Produções Intelectuais. Classes para realizar acesso ao banco de dados.

- Pacote `br.ufrn.sigaa.prodocente.lattes`: Composto por trinta e quatro (34) classes utilizadas para a importações de dados de produção intelectual da Plataforma *Lattes* para o SIGAA.

- Pacote `br.ufrn.sigaa.prodocente.producao.dominio`: Contém quarenta (40) classes de domínio referentes às produções do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.producao.jsf`: Composto por quarenta e duas (42) classes *Managed Beans* referentes ao gerenciamento de produções do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.producao.jsf.dominio`: Contém uma (1) classe de domínio auxiliar utilizada nas operações de gerenciamento de produções do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.atividades.dominio`: Composto por quarenta e quatro (44) classes de domínio referentes a atividades docentes cadastradas no módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.atividades.jsf`: Contém trinta e oito (38) classes *Managed Beans* referentes a atividades docentes cadastradas no módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.atividades.negocio`: Composto por uma (1) classe de negócio referente a atividades docentes cadastradas no módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.jsf`: Contém nove (9) classes *Managed Beans* referentes a operações gerais do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.struts`: Composto por uma (1) classe *Actions Struts* referentes a operações gerais do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.relatorios.jsf`: Contém doze (12) classes *Managed Beans* utilizadas para a emissão de relatórios de produtividade do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.relatorios.dominio`: Composto por vinte e quatro (24) classes de domínio utilizadas para a emissão de relatórios de produtividade do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.ajax`: Contém uma (1) classe. Servlet utilizado em consultas com *Ajax* a dados do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.negocio`: Composto por treze (13) classes de negócio do módulo de Produção Intelectual de Docentes.

- Pacote `br.ufrn.sigaa.prodocente.negocio.relatorioprodutividade`: Contém sete (7) classes auxiliares (*processador e helper*) para emissão do relatório de produtividade de docentes.

- Pacote `br.ufrn.sigaa.prodocente.negocio.relatorioprodutividade.impl`: Composto por cento e onze (111) classes que implementam as regras de negócio para emissão do relatório de produtividade de docentes.

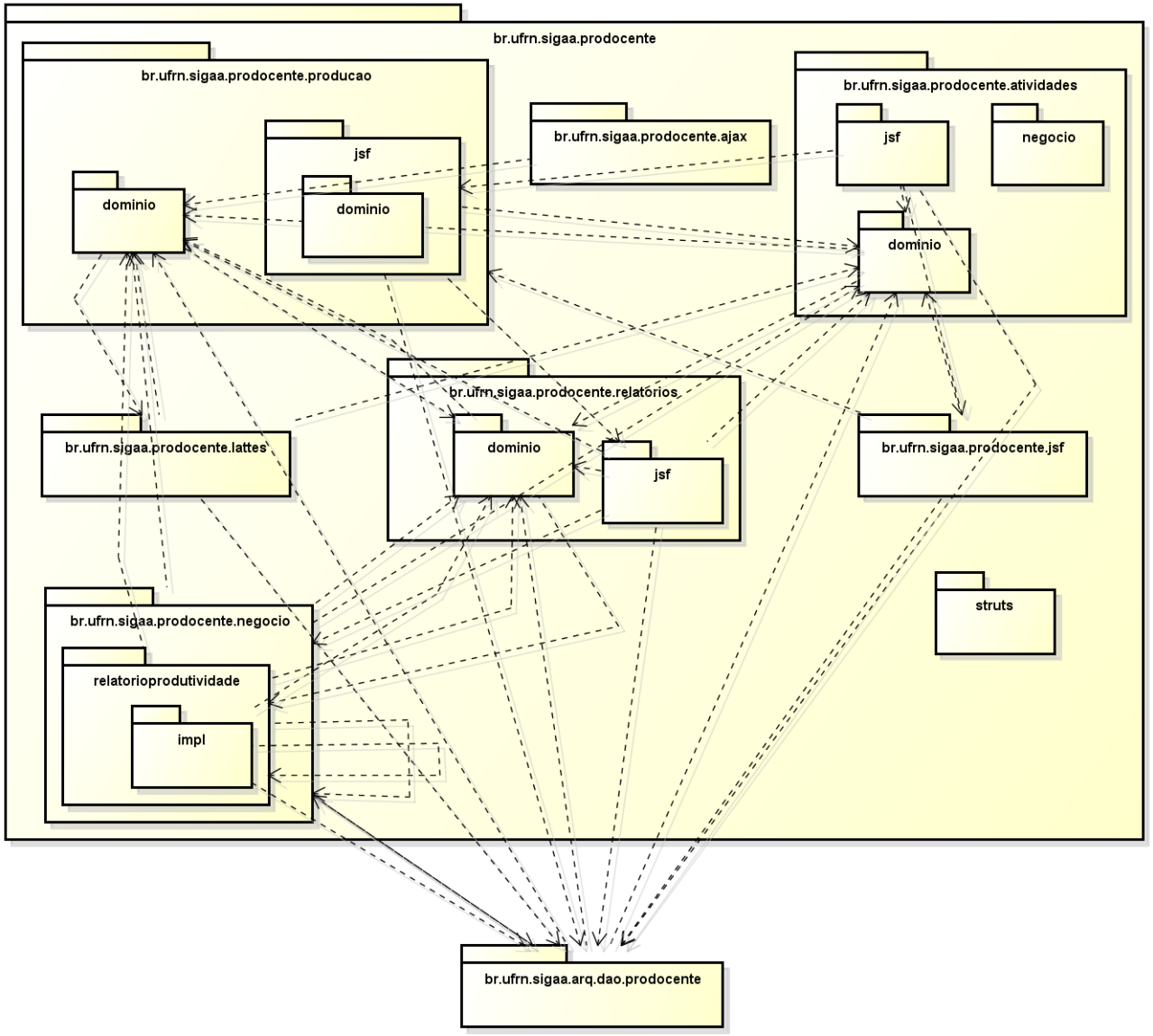


Figura 5.1: Diagrama de pacotes do módulo de Produção Intelectual

5.2 Levantamento das variações no módulo de Produção Intelectual

As classes que compõem o módulo de Produção Intelectual foram analisadas com base nas atividades de **manutenção** e **implementação** de novas funcionalidades, definidas pela UFS no sistema de Gerenciamento de Projetos *Redmine* [1]. Nesta seção, são apresentados os resultados do levantamento realizado, bem como uma análise das alterações e variações encontradas no módulo de Produção Intelectual a ser utilizada como base para apoiar as próximas fases deste estudo.

5.2.1 Variações encontradas

No *RedMine* foi realizada uma busca por todas as tarefas concluídas (situação atual fechada) e por tarefas do tipo manutenção ou inserção de novas funcionalidades. Com esses filtros aplicados, foram encontradas 285 tarefas. Cada uma dessas tarefas foi analisada, de modo a ser possível classificar cada alteração realizada. Alterações realizadas em esquemas e consultas de bancos de dados e alterações realizadas em páginas JSP não foram analisadas, pois não fazem parte do escopo deste trabalho. Após a análise, um total de trezentos e dez (310) alterações foram encontradas.

Após o levantamento das alterações e tabulação das informações, foi utilizado o *software WinMerge* para realizar uma análise das alterações encontradas. O *WinMerge* é uma ferramenta de código aberto para diferenciação e mesclagem de arquivos no sistema operacional *Windows*, possibilitando a comparação de pastas e arquivos e apresentando as diferenças em um formato visual de texto que facilita a compreensão e manipulação dos arquivos analisados.

Os arquivos alterados pela equipe de desenvolvimento da UFS foram comparados aos arquivos de código-fonte de uma versão não modificada do SIGAA. Essa comparação foi realizada com a finalidade de detectar e classificar as alterações realizadas de acordo com um catálogo de variações. Um total de noventa e seis (96) arquivos de classes Java foram

[1] `https://redmine.ufs.br/`

analisados com o *Winmerge*. A comparação foi realizada apenas no módulo de **Produção Intelectual**, entre a versão 3.5.15 do SIGAA modificada pela UFS e uma versão não modificada do código-base da UFRN, que também foi fornecida pela UFS. Após essa análise, as variações foram classificadas de acordo com o catálogo de variações apresentado na tabela 5.1 e discutido em detalhes na próxima seção.

5.2.2 Análise e classificação das variações

As alterações inseridas pela UFS módulo de produção intelectual foram classificadas e agrupadas de acordo com os tipos de variações apresentados na tabela 5.1. A análise dos tipos de variações permitiu a classificação de cada uma das trezentas e onze (311) variações encontradas, entre os vinte e quatro (24) tipos de variações apresentados na tabela 5.1. Na quarta coluna dessa tabela encontra-se o quantitativo de vezes em que cada variação foi inserida no módulo.

Para realizar a classificação, foi usado como base o catálogo de variações apresentado por Passos e Neto (2013). Além dos tipos de variações definidos por Passos e Neto (2013), foram encontrados 11 novos tipos de variações apresentados na tabela 5.1. Os seguintes novos tipos encontrados foram incluídos: *Adicionar constante, Adicionar construtor, Adicionar anotação, Adicionar controle de exceções, Modificar anotação, Modificar tipo de atributo, Modificar tipo de retorno do método, Modificar tipo genérico, Excluir comando condicional, Excluir atributo e Excluir chamada de método*. Além disso, 2 tipos catalogados por Passos e Neto (2013) foram excluídos: *Modificar valor do Array e Adicionar comando de repetição*, pois não foram encontradas variações desse tipo no módulo de **Produção Intelectual**.

Tabela 5.1: Tipos de variações encontradas no módulo de Produção Intelectual do SIGAA

Sigla	**Variação**	**Descrição da variação**	**Quantidade**
AC	Adicionar classe	Adicionar uma nova classe ao projeto	15
AM	Adicionar método	Adicionar um novo método	51
AA	Adicionar atributo	Adicionar um novo atributo	12
ACT	Adicionar constante	Adicionar uma nova constante	6
ACO	Adicionar construtor	Adicionar um novo construtor	1
AAN	Adicionar anotação	Adicionar uma anotação em atributo ou método	2
APR	Adicionar parâmetro	Adicionar um novo parâmetro a método	1
ACM	Adicionar chamada a método	Adicionar uma nova chamada de método	33
ACC	Adicionar comando condicional	Adicionar um comando condicional (*if*, *switch*)	61
ACE	Adicionar controle de exceções	Adicionar controle de exceções (*try*, *catch*)	2
MA	Modificar anotação	Modificar uma anotação de atributo ou método	2
MTA	Modificar tipo de atributo	Modificar o tipo do atributo	1
MVA	Modificar valor de atributo	Modificar o valor do atributo	3
MPR	Modificar parâmetro	Modificar o parâmetro de um método	16
MCM	Modificar chamada a método	Modificar uma chamada de método	41
MTG	Modificar tipo genérico	Modificar um tipo genérico `<T>`	1
MLS	Modificar literal *String*	Modificar uma literal *string* de um método que tem como tipo de retorno uma *String*	5
MEC	Modificar expressão condicional	Modificar uma expressão condicional (*if*, *switch*) de um método, alterar ou inserir	15
MIS	Modificar instrução SQL	Modificar uma instrução SQL (*select*, *insert*, *update*, *delete*) definida em um método	13
MTM	Modificar totalmente método	Modificar totalmente a implementação de um método	3
EM	Excluir método	Excluir método de classe	2
ECH	Excluir chamada de método	Excluir a chamada para um método	9
ECC	Excluir comando condicional	Excluir um comando condicional (*if*, *switch*)	3
EA	Excluir atributo	Excluir atributo de classe	1
		Total	**311**

Após a análise e classificação das alterações, foram elaborados os gráficos apresentados na figura 5.2 que apresenta a quantificação das ocorrências dos tipos de variações encontradas no módulo de **Produção Intelectual**.

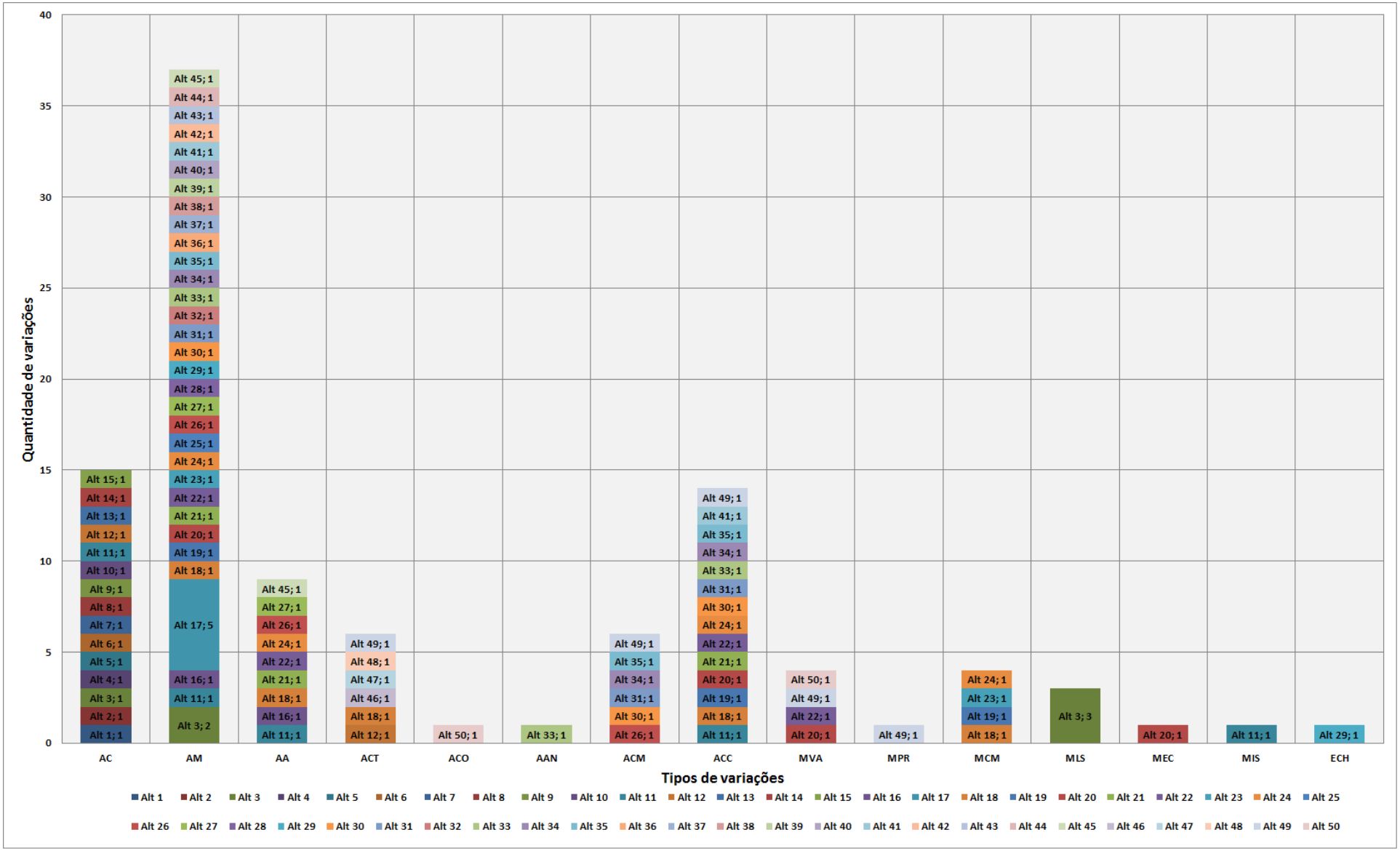
Devido ao grande volume de informações, o gráfico foi dividido em quatro partes, para melhor compreensão, porém cada figura apresenta as mesmas informações. No eixo vertical, foi colocada a **quantidade de variações encontradas** no módulo de **Produção Intelectual** e no eixo horizontal, **os tipos de variações catalogadas**, de acordo com as siglas da tabela 5.1. Nas colunas são apresentadas as alterações referentes a cada tipo de variação e a quantidade de ocorrências de cada tipo de variação encontrada.

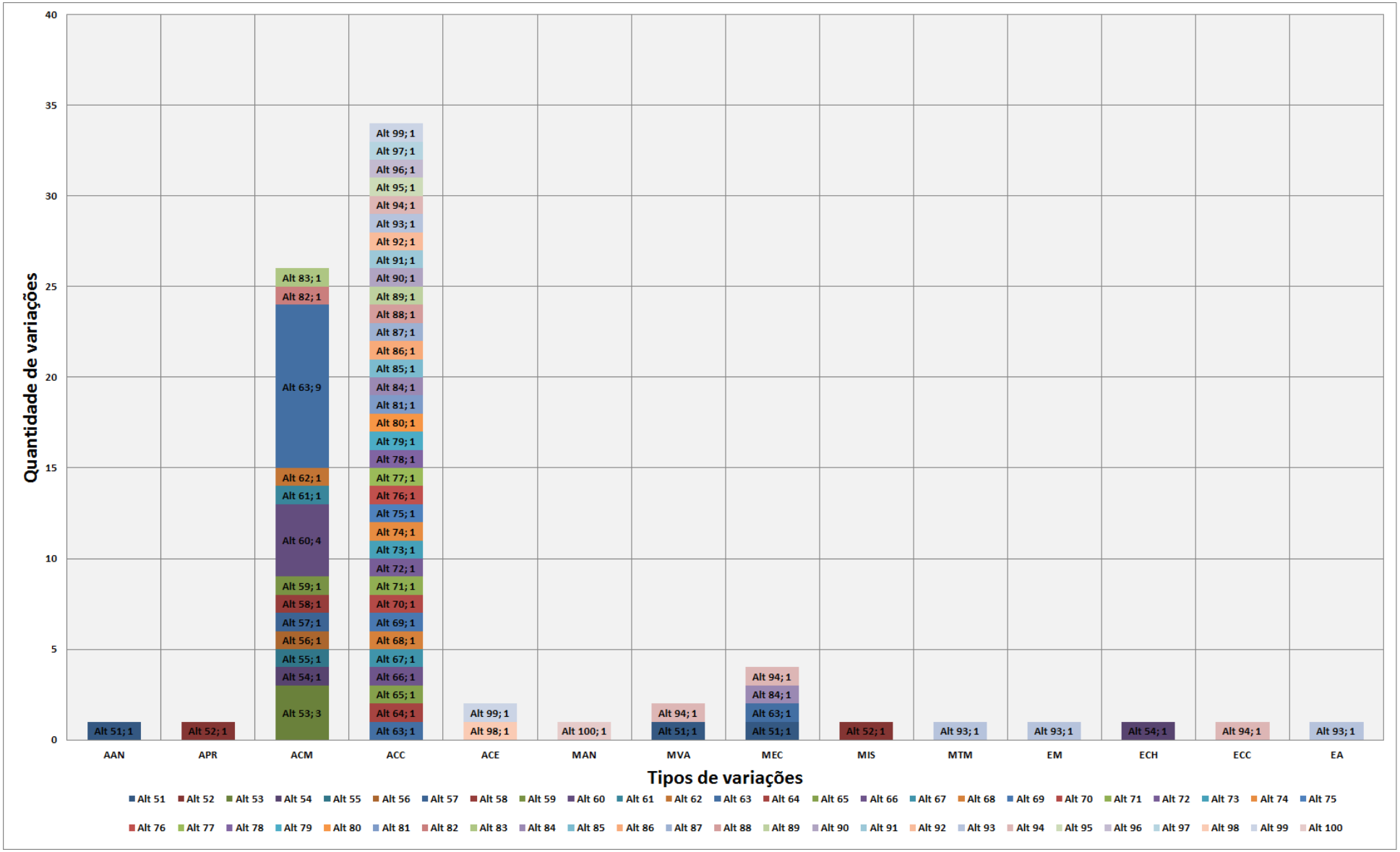
No primeiro gráfico da figura 5.2, é possível observar que algumas variações apresentam um número de ocorrências maior que um (1), isso ocorre pois são alterações *crosscutting*, ou seja, alterações que estão espalhadas por vários pontos do código fonte. Esse tipo de ocorrência foi registrada para as variações do tipo AM e ACM. Dentre as trezentas e onze (311) alterações analisadas, foram encontradas cinquenta (50) ocorrências de variações *crosscutting*, a "Alt 17" se repete cinco (5) vezes, a "Alt 63" se repete nove (9) vezes, a "Alt 60" se repete quatro (4) vezes, a "Alt 53" se repete três (3) vezes, a "Alt 139" se repete três (3) vezes, a "Alt 142" se repete cinco (5) vezes e a "Alt 188" se repete vinte e uma (21) vezes. Nesse tipo de ocorrência, as mudanças introduzidas pela UFS foram realizadas por meio da duplicação de código. Isso significa que o mesmo código duplicado foi introduzido 21 vezes em diferentes classes no módulo de produção intelectual. O processo de desenvolvimento adotado pela UFS, possibilita a introdução desse tipo de problema, como a repetição e o espalhamento do código.

É interessante destacar que, no segundo gráfico da figura 5.2, a alteração "Alt 65" classificada com o tipo de variação *ACM* é diferente da "Alt 65" classificada como *ACC*, assim como ambas são diferentes de outras ocorrências desta alteração classificadas em tipos de variações diferentes. Isso ocorre, pois para cada atividade de mudança realizada pela UFS, várias classes diferentes foram modificadas, introduzindo-se variações de tipos diferentes em pontos espalhados pelo código.

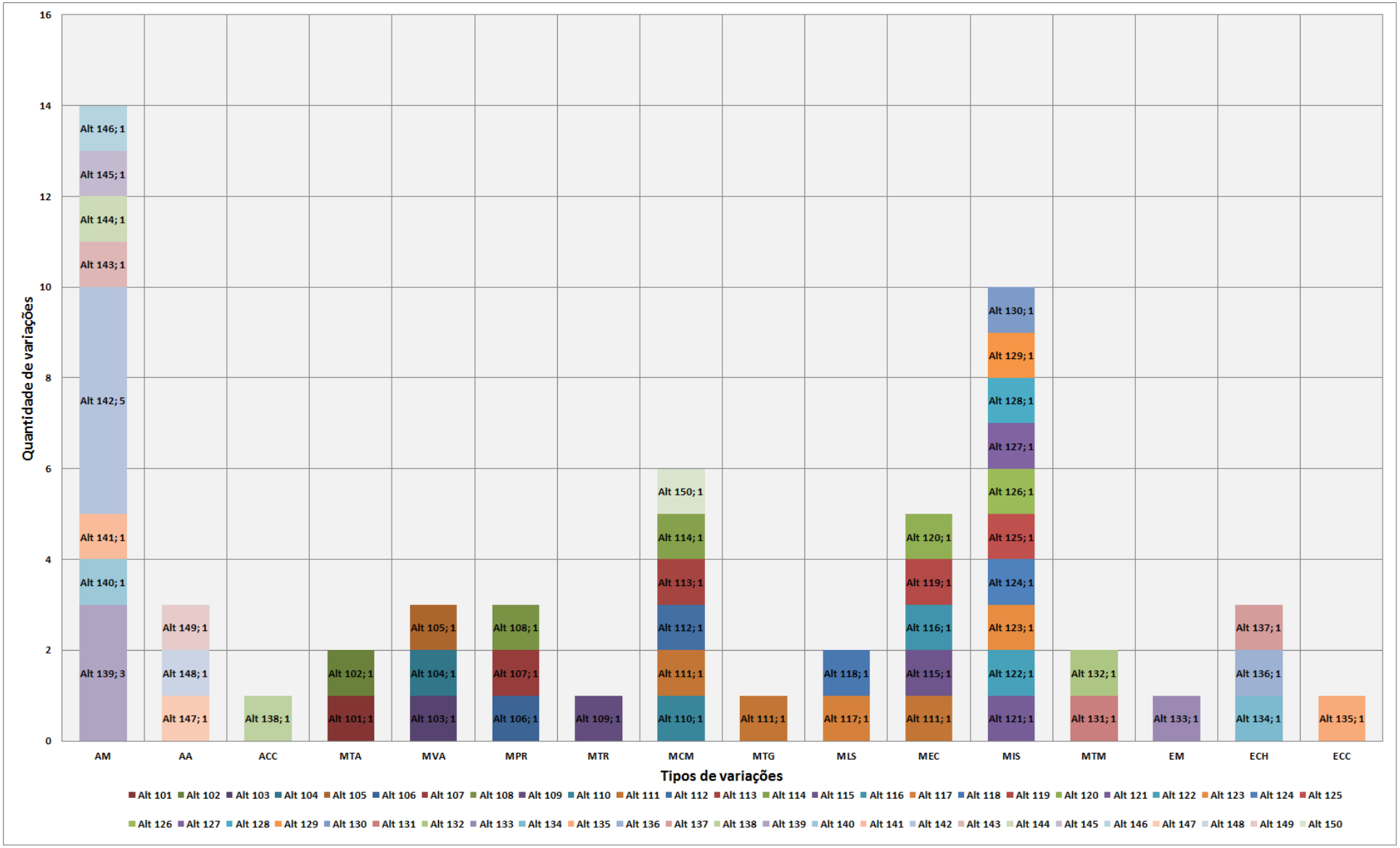
Figura 5.2: Quantificação das ocorrências dos tipos de variações no módulo de Produção Intelectual do SIGAA

(a) Alterações - gráf. 1

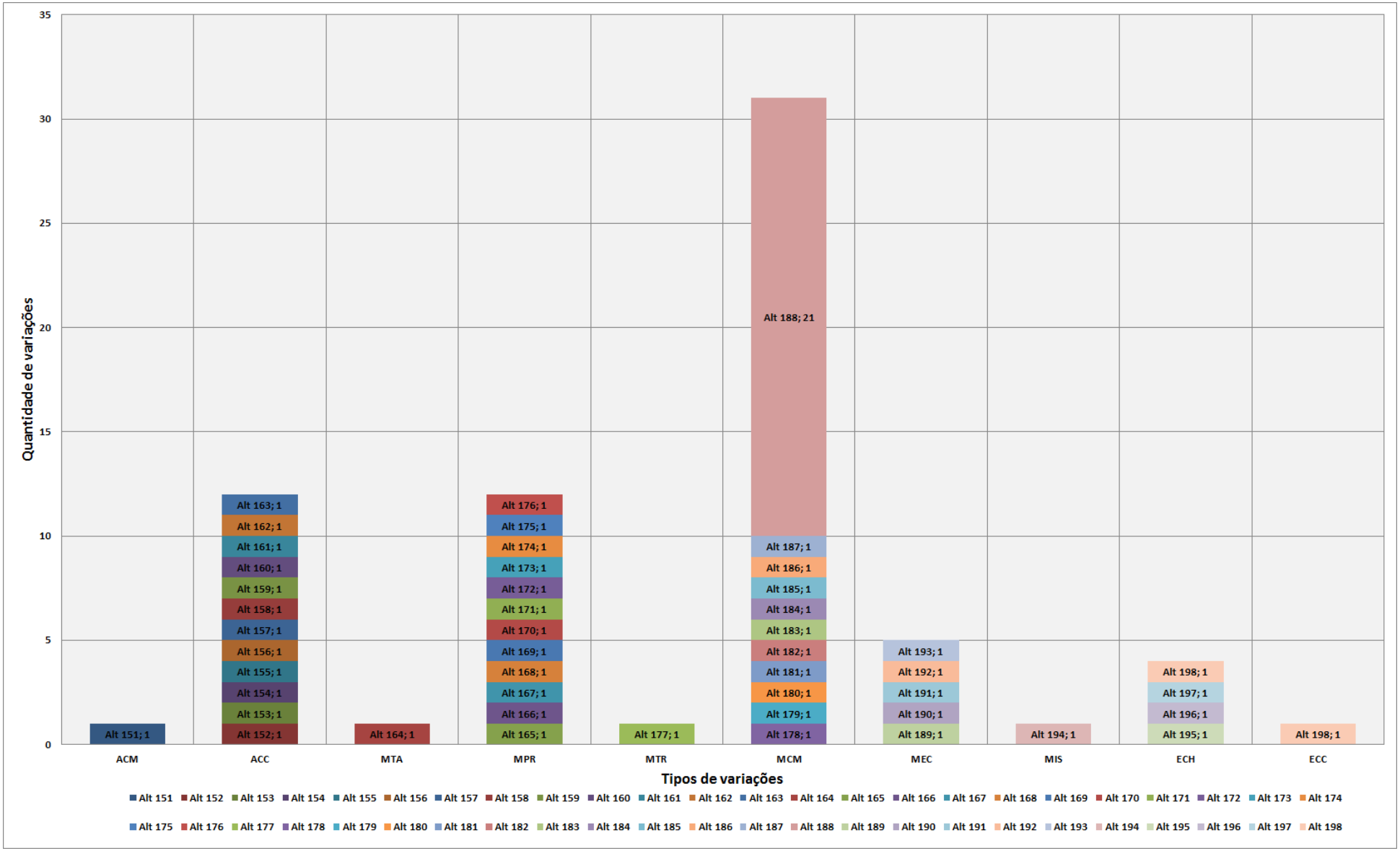




(a) Alterações - gráf. 2



(a) Alterações - gráf. 3



(a) Alterações - gráf. 4

5.3 Implementação das variações em AspectJ

Após realizar o levantamento e análise das alterações desenvolvidas pela UFS, foi iniciado o processo de implementação das variações utilizando AspectJ. Nesta seção, serão apresentadas as atividades executadas e os resultados obtidos com a implementação dos Aspectos.

Para executar as atividades desta etapa da pesquisa, foram utilizadas as seguintes ferramentas:

- Eclipse IDE (versão 4.4 - Luna [2]).
- *AspectJ Development Tools* (AJDT) - versão 2.2.4 [3], *plugin* para desenvolvimento em AspectJ, utilizando o Eclipse.
- Notepad++ [4]: programa para edição de código-fonte no sistema operacional Windows.

As variações foram separadas e implementadas em 109 Aspectos que foram nomeados de acordo as alterações realizadas. Esses Aspectos, que que contêm as variações, foram incluídos no pacote <`prodocente.br.ufs.sigaa.prodocente.aspectos`>.

Devido ao tamanho e à quantidade de alterações, o código não será apresentado integralmente neste trabalho. Porém, para que sejam discutidos alguns detalhes da implementação realizada, serão apresentados na próxima seção, partes dos Aspectos, separados de acordo com os tipos de variações implementadas.

5.3.1 Implementação de variações *Adicionar Método* e *Adicionar Atributo*

As variações do tipo *Adicionar Método* e *Adicionar Atributo* foram encontradas em pontos do código nos quais foram adicionados novos métodos e novos atributos a uma classe. Um exemplo desse tipo de variação, foi implementado na "Alt 2", *Ticket 11526*, que adiciona a exibição de dados do docente externo no relatório de produtividade, incluindo o novo atributo

[2] https://www.eclipse.org/luna/
[3] http://download.eclipse.org/tools/ajdt/44/dev/update
[4] https://notepad-plus-plus.org/

`private DocenteExterno docenteExternoRelatorio` e métodos `get` e `set` na classe `RelatorioProdutividadeMBean`.

Esses dois tipos de variações foram implementados com declarações *inter-type*, que permitem inserir membros como campos, métodos e construtores em uma classe. Foi implementado o Aspecto `InjetorDadosDocenteExterno` no qual foram incluídas declarações *inter-type* que modificam a classe `RelatorioProdutividadeMBean`, adicionando o atributo e os métodos. Na listagem 5.1, é apresentado o código do Aspecto `InjetorDadosDocenteExterno`. É possível observar que na linha 1 foi adicionado o atributo `docenteExternoRelatorio`, e nas linhas 2 e 5 foram adicionados os métodos `get` e `set` na classe `RelatorioProdutividadeMBean`, com o uso de declarações *inter-type*.

```
1  private DocenteExterno RelatorioProdutividadeMBean.
       docenteExternoRelatorio;
2  public DocenteExterno RelatorioProdutividadeMBean.
       getDocenteExternoRelatorio() {
3       return docenteExternoRelatorio;
4  }
5  public void RelatorioProdutividadeMBean.
       setDocenteExternoRelatorio(
6                   DocenteExterno docenteExternoRelatorio) {
7                     this.docenteExternoRelatorio =
                          docenteExternoRelatorio;
8  }
```

Listagem 5.1: Implementação de variações *Adicionar Método* e *Adicionar Campo* com AspectJ

Com AspectJ, foi possível implementar esses dois tipos de variações, com declarações *inter-type*, sem modificar diretamente a classe `RelatorioProdutividadeMBean`.

5.3.2 Implementação de variações *Adicionar chamada a método*, *Adicionar comando condicional* e *Excluir chamada de método*

Um exemplo desses três tipos de variações ocorreu nas alterações realizadas no método `validate` da classe `AudioVisual`. As alterações realizadas nesse método são apresentadas na figura 5.3, na qual é possível comparar o código original 5.3a e o código modificado pela UFS 5.3b.

Nesse método foram introduzidas variações dos três tipos: *Adicionar chamada a método*, *Adicionar comando condicional* e *Excluir chamada de método*. Devido à quantidade de modificações realizas, a melhor abordagem encontrada para implementar essas alterações com AspectJ foi interceptar a execução do método e utilizar um `advice` do tipo `around` que executa o código modificado no lugar do método `validate` original.

```
@Override
public ListaMensagens validate(){

    ListaMensagens lista = new ListaMensagens();

    ValidatorUtil.validateRequired(getTitulo(), "Título", lista);
    ValidatorUtil.validateRequired(getAutores(),"Autores", lista);
    ValidatorUtil.validateRequired(getLocal(),"Local", lista);
    if(getTipoParticipacao()!=null)
        ValidatorUtil.validateRequiredId(getTipoParticipacao().getId(),"Tipo de Participação",
lista);
    else
        ValidatorUtil.validateRequired(getTipoParticipacao(),"Tipo de Participação", lista);
    if(getTipoRegiao()!=null)
        ValidatorUtil.validateRequiredId(getTipoRegiao().getId(), "Âmbito", lista);
    else
        ValidatorUtil.validateRequired(getTipoRegiao(), "Âmbito", lista);
    if(getTipoArtistico()!=null)
        ValidatorUtil.validateRequiredId(getTipoArtistico().getId(), "Tipo Artístico", lista);
    else
        ValidatorUtil.validateRequired(getTipoArtistico(), "Tipo Artístico", lista);
    if(getSubTipoArtistico()!=null)
        ValidatorUtil.validateRequiredId(getSubTipoArtistico().getId(), "Classificação", lista);
    else
        ValidatorUtil.validateRequired(getSubTipoArtistico(), "Classificação", lista);
    ValidatorUtil.validateRequired(getDataProducao(), "Data de Produção", lista);
    ValidatorUtil.validateRequired(getAnoReferencia(), "Ano de Referência", lista);

    return lista;
}
```

(a) Código original

```
@Override
public ListaMensagens validate(){

    ListaMensagens lista = new ListaMensagens();

    //MUDANCA Ticket #8514 Remoção de validação dupla

    //MUDANCA Ticket #7972 Não permitir valores negativos
    if(getDuracaoDivulgacao()!=null && !(getDuracaoDivulgacao() > 0)){
        lista.addErro("Duração: Valor Inválido!");

    }
    //MUDANCA Ticket #7790 validação do ano de referência com a data de produção
    lista.addAll(super.validate().getMensagens());

    return lista;
}
```

(b) Código modificado

Figura 5.3: Comparação entre o código original do método `validate` da classe `AudioVisual` e o código modificado pela UFS

No Aspecto implementado, foram incluídas as alterações referentes aos *tickets* 8514 (“Alt 137”) , 7972 (“Alt 95”) e 7790 (“Alt 63”). A “Alt 137” introduz a variação do tipo *Excluir chamada de método* para remover a validação dupla de dados. A “Alt 95“ introduziu a validação de dados negativos, por meio da inclusão de uma variação do tipo *Adicionar comando condicional* e a “Alt 63” incluiu uma chamada ao método da superclasse que realiza a validação entre ano de referência e ano de uma produção (uma variação do tipo *Adicionar chamada a método*). Na listagem 5.2, é apresentado o código fonte do Aspecto `CorrecaoValidacaoAudioVisual` que contém essas alterações.

```
1  ListaMensagens around(AudioVisual audioVisual):
2    target(audioVisual)
3    && execution(public ListaMensagens AudioVisual.validate())
4    {
5      ListaMensagens lista = new ListaMensagens();
6      //MUDANCA Ticket #8514
7      //MUDANCA Ticket #7972
```

```
8         if(audioVisual.getDuracaoDivulgacao()!=null
9             && !(audioVisual.getDuracaoDivulgacao() > 0)){
10          lista.addErro("Duração: Valor Inválido!");
11        }
12        //MUDANCA Ticket #7790
13        lista.addAll(audioVisual.callSuperValidate().getMensagens());
14        return lista;
15    }
```

Listagem 5.2: Implementação de variações *Adicionar chamada a método Adicionar comando condicional* e *Excluir chamada de método* com AspectJ

Para implementar a "Alt 63" na classe `AudioVisual`, foi necessário realizar uma adaptação no código introduzido pela UFS, devido a uma limitação da linguagem AspectJ. Na alteração, apresentada na figura 5.3b, é possível observar que foi introduzida uma chamada ao método `validate` da superclasse de `AudioVisual`. Contudo, em um `advice` não é possível realizar uma chamada do tipo `super` pois o `advice` não está vinculado a nenhuma classe específica. Para implementar essa alteração, foi preciso adotar o seguinte artifício: o método `callSuperValidate()` foi implementado e adicionado à classe `AudioVisual` com uma declaração *inter-type*. Nesse método, apresentado na listagem 5.3, é realizada apenas uma chamada ao método `super.validate()`, isto é, ao método `validate()` da classe `ProducaoArtisticaLiterariaVisual`, a superclasse de `AudioVisual`. Após adicionar essa declaração *inter-type*, foi adicionada a chamada ao método `callSuperValidate`, na implementação da "Alt 63", apresentada na linha 14 da listagem 5.2. Como o método `callSuperValidate` é privado, seu escopo está limitado apenas ao Aspecto no qual foi implementado.

```
1  private ListaMensagens AudioVisual.callSuperValidate(){
2    return super.validate();
3  }
```

Listagem 5.3: Adição do método `callSuperValidate()` para resolver problema na implementação da "Alt 63"

Esse problema também ocorreu na implementação da "Alt 63" em outras classes (`BolsaObtida`,

`ParticipacaoSociedade`, `ProducaoArtisticaLiterariaVisual`, `Publicacao`, `ParticipacaoColegiadoComissao`, `PremioRecebido`, `VisitaCientifica` e `ParticipacaoComissaoOrgEventos`). Em todos os casos, foi adotado o mesmo artifício.

5.3.3 Implementação de variação *Adicionar construtor*

A variação do tipo *Adicionar construtor* foi introduzida apenas uma vez no módulo de Produção Intelectual. Na "Alt 50", *ticket* 12311, foi adicionado um novo construtor para a classe `Artigo`. O novo construtor foi adicionado com uma declaração *inter-type*. O código dessa implementação é apresentado na listagem 5.4. Essa alteração foi implementada no Aspecto `InjetorNovoConstrutorArtigo`.

```
1  public aspect InjetorNovoConstrutorArtigo {
2  /*...*/
3     public Artigo.new(int id, String titulo,
4           String tipoParticipacao, Integer ano,
5           Boolean validado, Integer idArquivo,
6           Integer sequenciaProducao,
7           String tipoPeriodico, String issn ) {
8          this();
9          setId(id);
10         setTitulo(titulo);
11         TipoParticipacao tipo = new TipoParticipacao();
12         tipo.setDescricao(tipoParticipacao);
13         setTipoParticipacao(tipo);
14         ...
15    }
16 }
```

Listagem 5.4: Exemplo de implementação de variação *Adicionar construtor*

Durante a implementação dessa variação, foi encontrada uma limitação do AspectJ. Apesar de ter sido feita a inicialização das variáveis e a adição do construtor com uma declaração

inter-type que estava de acordo com o manual da linguagem [5], o compilador do AJDT, estava fornecendo o seguinte aviso: *inter-type constructor does not contain explicit constructor call: field initializers in the target type will not be executed [Xlint:noExplicitConstructorCall]*. Este aviso indicava que o construtor não estava sendo adicionado à classe Artigo como deveria, pois não havia sido declarada uma chamada explícita a um construtor da classe. Para solucionar esse problema, foi adicionada a instrução `this()` (linha 8 da listagem 5.4), que realiza uma chamada a outro construtor da classe Artigo. Pelo comportamento observado, é necessário adicionar uma chamada explícita a outro construtor da classe para que os campos sejam inicializados corretamente e o construtor declarado seja adicionado [6]. Após essa modificação, a declaração *inter-type* foi adicionada corretamente à classe Artigo.

5.3.4 Implementação de variação *Adicionar constante*

Em alguns pontos do código do módulo de Produção Intelectual foram adicionadas novas constantes, esse tipo de variação foi implementada com declarações *inter-type*. Um exemplo é a "Alt 18", `ticket` 20690, na qual foi adicionada a constante DIRETOR_MEDIO à classe `SigaaPapeis`. Essa alteração é apresentada na listagem 5.5, a implementação foi extraída do Aspecto `InclusaoOperacoesDeChefe`. As variações do tipo *Adicionar constante* foram implementadas utilizando esse tipo de declaração, exceto no caso da constante SISTEMA_EM_PROCESSAMENTO que foi inserida na interface `ParametrosGraduacao`. Essa alteração foi adaptada, pois não foi possível adicionar uma constante em uma interface com uma declaração *inter-type*. Por isso, foi adicionada uma constante local no Aspecto `InclusaoOperacoesDeChefe` contendo o mesmo valor da constante, que posteriormente foi utilizada em outra alteração também realizada nesse Aspecto.

```
1     public static final int SigaaPapeis.DIRETOR_MEDIO =
2           SigaaSubsistemas.MEDIO.getId() + 7;
```

Listagem 5.5: Implementação de variação *Adicionar constante*

[5] https://eclipse.org/aspectj/doc/released/progguide/language-interType.html

[6] http://stackoverflow.com/questions/17647587

5.3.5 Implementação de variação *Adicionar anotação*

Esse tipo de variação foi encontrada em pontos do código nos quais foram inseridas novas anotações (em métodos, atributos e classes). Para implementar essa variação, foi utilizado um recurso do AspectJ 5: a declaração de anotação. Esse recurso permite inserir anotações em classes, atributos, métodos e construtores, utilizando respectivamente as instruções `declare @type`, `declare @field`, `declare @method` e `declare @constructor`. A "Alt 51", *ticket* 8442, é um exemplo desse tipo de variação, na qual foi incluída a anotação `@JoinColumn` no campo `disciplinaQualificacao` da classe `QualificacaoDocente`. Com a instrução `declare @field` foi inserida a anotação no campo `disciplinaQualificacao`. A implementação da "Alt 51" é apresentada na listagem 5.6.

```
1  declare @field: private Collection<DisciplinaQualificacao>
       QualificacaoDocente.disciplinaQualificacao: @JoinColumn(name =
        "id_qualificacao", unique = false, nullable = true,
       insertable = true, updatable = true) ;
```

Listagem 5.6: Implementação de variação *Adicionar anotação*

Todas as alterações desse tipo foram implementadas com a declaração de anotação.

5.3.6 Implementação das variações *Adicionar parâmetro* e *Modificar parâmetro*

Esse tipo de variação foi introduzida no SIGAA, com a alteração de métodos existentes: adicionando um parâmetro a um método ou alterando o tipo de um parâmetro de métodos. Um exemplo de variação do tipo *Adicionar parâmetro real* ocorre na "Alt 52", *ticket* 8446, na qual foi adicionado o parâmetro `paginacao` ao método `findAllUnidade` da classe `ChefiaDao`. Na figura 5.4, é apresentado o método original 5.4a e o método modificado pela UFS 5.4b.

```
    public Collection<?> findAllUnidade(Class<?> atividade, Unidade unidade) throws
DAOException {
```

(a) Código original

```
   //MUDANCA Ticket #8446 adicionando paginação

   public Collection<?> findAllUnidade(Class<?> atividade, Unidade unidade,
PagingInformation paginacao) throws DAOException {
```

(b) Código modificado

Figura 5.4: Comparação entre o código original do método `findAllUnidade` da classe `ChefiaDao` e o código modificado pela UFS

Para implementar as variações desse tipo foram utilizadas construções *inter-type* com o objetivo de adicionar os métodos com novos parâmetros às classes originais. Esta opção permitiu adicionar os novos métodos sem modificar os métodos implementados originalmente.

A listagem 5.7 apresenta a implementação da "Alt 52", na qual foi adicionado o método `findAllUnidade` à classe `ChefiaDao`, com o novo parâmetro `paginacao`. Essa listagem foi extraída do Aspecto `AddPaginacaoChefia`.

```
1  public Collection<?> ChefiaDao.findAllUnidade(Class<?> atividade,
2                          Unidade unidade, PagingInformation
                              paginacao)
3                          throws DAOException {
4      StringBuilder hql = new StringBuilder();
5      ...
6      //MUDANCA Ticket #8446 adicionando paginação
7      Query q = getSession().createQuery(hql.toString());
8      if (paginacao != null) {
9        paginacao.setTotalRegistros(q.list().size());
10       q.setFirstResult(paginacao.getPaginaAtual() * paginacao.
             getTamanhoPagina());
11       q.setMaxResults(paginacao.getTamanhoPagina());
12     }
13     return q.list();
```

```
14      }
```

Listagem 5.7: Implementação da variação *Adicionar parâmetro*

A mesma estratégia foi adotada para incluir variações do tipo *Modificar parâmetro*. Esse tipo de variação foi considerada em pontos do código nos quais um parâmetro foi modificado. A "Alt 165", *ticket 11091*, é um exemplo desse tipo de variação. Nessa alteração, o parâmetro `servidor` do tipo `Servidor` foi trocado pelo parâmetro `docente` do tipo `Object`. Na listagem 5.8, é apresentada a implementação do método `findByProducaoServidor`, que foi extraída do Aspecto `AdaptacaoMPIDocenteExterno`, com a alteração no parâmetro.

```
1  public Collection<Producao> ProducaoDao.findByProducaoServidor(
2      Object docente, Class tipo, PagingInformation paginacao)
3      throws DAOException {
4    Query qTotal = getSession().createQuery(
5        "select count(p.id) from " + tipo.getSimpleName() + " p
             where "
6        + ((docente instanceof Servidor) ? "p.servidor.id = "
7    ...
8    return c.list();
9  }
```

Listagem 5.8: Exemplo de variação do tipo *Modificar parâmetro*

5.3.7 Implementação de variação *Modificar valor de atributo*

Esse tipo de variação foi encontrada em pontos do código nos quais o valor de atributos foram alterados, como no caso da "Alt 105", *ticket* 8372. Nessa alteração, o valor do atributo `anoVigencia` da classe `RelatorioProdutividadeMBean` foi alterado do ano anterior para o ano atual. Na figura 5.5, é possível observar o código da classe `RelatorioProdutividadeMBean` antes (5.5a) e depois (5.5b) da alteração.

```
private Integer anoVigencia = CalendarUtils.getAnoAtual() - 1;
```

(a) Código original

```
//MUDANCA Ticket #8372 Tornar ano de vigencia o atual
private Integer anoVigencia = CalendarUtils.getAnoAtual();
```

(b) Código modificado

Figura 5.5: Alteração realizada no atributo `anoVigencia` da `RelatorioProdutividadeMBean`

Para implementar essa alteração, foi utilizado um *advice* do tipo `after` e um *pointcut* para capturar a execução do construtor da classe `RelatorioProdutividadeMBean`, à qual pertence o atributo e foi inserido o código que altera o valor do atributo para o ano atual. Na listagem 5.9 é apresentado o Aspecto `CorrecaoRelatorioProdMBeanAnoAtual` em que foi implementada esta alteração.

```
1  public aspect CorrecaoRelatorioProdMBeanAnoAtual {
2    after(RelatorioProdutividadeMBean rpMBean):
3    execution( public RelatorioProdutividadeMBean.new() )
4    && target(rpMBean) {
5    rpMBean.setAnoVigencia(CalendarUtils.getAnoAtual());
6  }
7 }
```

Listagem 5.9: Exemplo de variação do tipo *Modificar valor de atributo*

Outras ocorrências desse tipo de variação foram implementadas com esse mesmo artifício, o que possibilitou a alteração de valor do atributo sempre após a classe ser instanciada e sem modificar a classe original.

5.3.8 Implementação de variação *Modificar chamada a método*

Variações desse tipo foram encontradas em pontos do código nos quais uma chamada a método foi alterada, em termos de argumentos ou substituída por uma chamada a outro método.

A “Alt 110”, *ticket* 20563, é um exemplo desse tipo de variação. Nessa alteração, a chamada ao método `findByTipoUnidadeAcademica` da classe `UnidadeMBean` foi modificada dentro do método `getAllDepartamentoUnidAcademica`, como é possível visualizar na figura 5.6.

```
   public Collection<Unidade> getAllDepartamentoUnidAcademica() throws
DAOException {

        return getDAO(UnidadeDao.class).findByTipoUnidadeAcademica(Tip
oUnidadeAcademica.DEPARTAMENTO,
                TipoUnidadeAcademica.UNID_ACADEMICA_ESPECIALIZADA);

    }
```

(a) Código original

```
   public Collection<Unidade> getAllDepartamentoUnidAcademica() throws
DAOException {
        //MUDANCA Manutenção #11116, #20563
        return getDAO(UnidadeDao.class).findByTipoUnidadeAcademica(Tip
oUnidadeAcademica.DEPARTAMENTO,
                TipoUnidadeAcademica.UNID_ACADEMICA_ESPECIALIZADA,TipoU
nidadeAcademica.PROGRAMA_POS, TipoUnidadeAcademica.ESCOLA);

    }
```

(b) Código modificado

Figura 5.6: Exemplo de variação do tipo *Modificar chamada a método*

Para substituir a chamada ao método original com dois argumentos pela chamada ao novo método com quatro argumentos, foi utilizado um `advice` do tipo `around` e um *pointcut* para capturar a execução do método `getAllDepartamentoUnidAcademica` (no qual foi realizada a modificação). A implementação dessa alteração, extraída do Aspecto `InjetorUnidadeCODAP`, é apresentada na listagem 5.10. A execução do método `getAllDepartamentoUnidAcademica` foi interceptada e substituída pelo código alterado.

```
1  Collection<Unidade> around (  UnidadeMBean mBean) throws
       DAOException:
2    execution(  public Collection<Unidade> UnidadeMBean.
         getAllDepartamentoUnidAcademica()  )
```

```
3     && target(mBean)
4     {
5       return mBean.getDAO(UnidadeDao.class).
            findByTipoUnidadeAcademica(
6         TipoUnidadeAcademica.DEPARTAMENTO,
7         TipoUnidadeAcademica.UNID_ACADEMICA_ESPECIALIZADA,
8         TipoUnidadeAcademica.PROGRAMA_POS, TipoUnidadeAcademica.
              ESCOLA);
9     }
```

Listagem 5.10: Exemplo de variação do tipo *Modificar chamada a método*

5.3.9 Implementação de variação *Modificar totalmente método*

Esse tipo de variação foi encontrada em pontos nos quais o código do método foi completamente alterado. A "Alt 132", *ticket* 8148, apresenta uma variação desse tipo, na qual o método `cadastrar` da classe `BolsaObtidaMBean` foi completamente alterado pela UFS. Na figura 5.7, é possível observar que o código original implementado pela UFRN (5.7a) foi completamente removido na alteração realizada pela UFS (5.7b).

```
public String cadastrar() throws SegurancaException, ArqException,
        NegocioException {

    montarData();
    if (hasErrors())
        return null;

    super.cadastrar();

    if (hasErrors()) {
        return null;
    } else {
        if (isPesquisa()) {
            return cancelar();
        } else {
            return preCadastrar();
        }
    }
}
```

(a) Código original

```
public String cadastrar() throws SegurancaException, ArqException,
        NegocioException {
    //MUDANCA Ticket #8148 limpando campos indevidamente ao validar
    return super.cadastrar();


}
```

(b) Código modificado

Figura 5.7: Alteração do código no método `cadastrar` da classe `BolsaObtidaMBean`

Para implementar este tipo de variação, também foi utilizado o `advice` do tipo `around`, e substituída a execução do código método original, pelo código modificado. O excerto do Aspecto `AlteraCadastroBolsaObtidaMBean` apresentado na listagem 5.11, contém a implementação da "Alt 132".

```
1  String around(BolsaObtidaMBean mBean) throws SegurancaException,
       ArqException,
2  NegocioException :
3    execution (public String BolsaObtidaMBean.cadastrar())
4  && target(mBean)
5  {
6    return mBean.superCadastrar();
7  }
8  private String BolsaObtidaMBean.superCadastrar() throws
       SegurancaException, ArqException,
9  NegocioException {
10   return super.cadastrar();
11 }
```

Listagem 5.11: Exemplo de variação do tipo *Modificar totalmente método*

Nessa alteração, também foi necessário introduzir uma chamada à superclasse com uma construção *inter-type*, o método `superCadastrar`. Esse artifício, apresentado anteriormente na listagem 5.3, foi usado pois não é possível realizar dentro de um `advice` uma chamada à superclasse.

5.3.10 Implementação de variação *Adicionar controle de exceções*

Alterações nas quais foi inserido no código fonte um bloco de instruções do tipo `try - catch` foram classificadas como variação *Adicionar controle de exceções* do tipo. Um exemplo é a "Alt 98", *ticket* 8496, na qual foi inserido um bloco `try - catch` no método `cancelar` da classe `TeseOrientadaMBean`. Na figura 5.8, é possível visualizar o código original do método `cancelar` 5.8a e o código modificado após a "Alt 98" 5.8b.

Na listagem 5.12, é apresentada a implementação da "Alt 98". Nessa alteração, extraída do Aspecto `CorrecaoDirecTeseOrientadaMBean`, foi utilizado um *advice* do tipo `around` e um `pointcut` para interceptar a execução código do método `cancelar`.

Outras alterações desse tipo também foram implementadas com essa abordagem.

```
@Override
public String cancelar() {
    initObj();
    return super.cancelar();
}
```

(a) Código original

```java
@Override
public String cancelar() {
    initObj();
    //MUDANCA: TICKET 8496:Direcionamento errado ao clicar em "Cancelar"
    //return super.cancelar();
    try {
        if (getIdTipoOrientacao()!= -1) {
          buscarTeses();
        }
        return forward(getListPage());
    } catch (DAOException e) {
        notifyError(e);
        e.printStackTrace();
    }
    return null;
}
```

(b) Código modificado

Figura 5.8: Alteração no método `cancelar` da classe `TeseOrientadaMBean`

```
1  String around(TeseOrientadaMBean mBean):
2    execution(public String TeseOrientadaMBean.cancelar())
3    && target(mBean)
4    {
5      try {
6        if (mBean.getIdTipoOrientacao()!= -1) {
7          mBean.buscarTeses();
8        }
9        return mBean.forward(mBean.getListPage());
10     } catch (DAOException e) {
11       mBean.notifyError(e);
12       e.printStackTrace();
13     }
14     return null;
15   }
```

Listagem 5.12: Implementação da "Alt 98"

5.3.11 Implementação de variação *Excluir atributo* e *Excluir método*

Esses tipos de variações foram implementados parcialmente, pois não há uma construção específica de AspectJ que possibilite a remoção de um atributo ou método de classe. Um exemplo desse tipo de variação ocorreu na "Alt 93", *ticket* 8146, na qual foram removidos da classe `VisitaCientificaMBean`, os atributos `private String dataInicio` e `private String dataFim`, os métodos *getters* e *setters* desses atributos e também o método `cadastrar`. A exclusão dos atributos não foi implementada. Porém, foi realizada a remoção de pontos do código da classe nos quais esses atributos eram utilizados. Para isso, foram usados *advice* do tipo `around` e alterados os métodos que utilizavam esses atributos. Na listagem 5.13, é apresentada uma alteração no método `clear` da classe `VisitaCientificaMBean` no qual eram utilizados os dois atributos.

```
1  void around(VisitaCientificaMBean vMBean):
2              execution(private void VisitaCientificaMBean.clear())
3              && target(vMBean)
4            {
5    vMBean.setObj(new VisitaCientifica());
6    vMBean.getObj().setTipoParticipacao(new TipoParticipacao(
       TipoParticipacao.AUTOR_GENERICO));
7    if (vMBean.getDocenteLogado() instanceof Servidor) {
8      vMBean.getObj().setDepartamento(vMBean.getServidorUsuario().
         getUnidade());
9    } else {
10     vMBean.getObj().getIes().setId(((DocenteExterno) vMBean.
         getDocenteLogado()).getInstituicao().getId());
11     vMBean.getObj().setDepartamento(null);
12   }
13 }
```

Listagem 5.13: Alteração no método `clear` da classe `VisitaCientificaMBean`

Os métodos `get` e `set` dos atributos também não foram removidos. Porém, foi necessário adaptar a remoção do método `cadastrar`, uma vez que este sobrescrevia o método

da superclasse. Para adaptar essa alteração, a implementação desse método foi substituída por uma chamada ao método da superclasse, utilizando-se um `advice` do tipo `around`. Na listagem 5.14, é apresentada a adaptação da variação do tipo *Excluir método*.

```
1  String around(VisitaCientificaMBean vMBean) throws
       SegurancaException, ArqException, NegocioException:
2    execution(private String VisitaCientificaMBean.
       cadastrar())
3    && target(vMBean)
4  {
5  return vMBean.callSuperCadastrar();
6
7  }
8  private String VisitaCientificaMBean.callSuperCadastrar
       () throws SegurancaException, ArqException,
       NegocioException
9  {
10   return super.cadastrar();
11 }
```

Listagem 5.14: Exemplo de variação do tipo *Excluir método* com adaptações

Esses dois tipos de variações foram implementados parcialmente com adaptações.

5.3.12 Implementação de variação *Modificar literal String*

Este tipo de variação foi observada em alterações nas quais foram modificados os valores de uma *String* em um método que tem como tipo de retorno uma *String*. Na "Alt 3", *ticket* 6721, por exemplo, foi modificada uma *String* de retorno no método `getTituloView` da classe `OrientacaoICExterno`. Abaixo é apresentado o código original (na figura 5.9a) do método e o código modificado (na figura 5.9b) após a alteração.

```
@Transient
public String getTituloView() {
      return  "    <td>Nome do Orientando</td><td>Tipo</td>";



}
```

(a) Código original

```
@Transient
public String getItemView() {
    //MUDANCA: Ticket #6721 Adição das colunas de data no relatório
    return "  <td>"+ getNomeOrientando()+ "</td>" +
            "<td style=\"text-align:center\">"+Formatador.getInstance().formatarData(dataInicio)+" -
+Formatador.getInstance().formatarData(dataFim)+
            //MUDANCA Ticket #11015 - retirar coluna tipo
            "</td>"
            //MUDANCA #20090 Modificações na exibição do relatório produtividade do docente
            //"<td>Externo</td>";
             +"<td align=\"center\"> - </td>";
}
```

(b) Código modificado

Figura 5.9: Alteração no método `getTituloView` da classe `OrientacaoICExterno`

Na listagem 5.15, é apresentada a implementação dessa alteração.

```
1            String around( ):
2            execution(public String OrientacaoICExterno.
                getTituloView())
3            {
4              return  "    <td>Nome do Orientando</td><td style=\"
                  text-align:center\">Período</td>";
5                 //"<td>Tipo</td>";
6            }
```

Listagem 5.15: Exemplo de variação do tipo *Modificar literal String*

As demais alterações desse tipo também foram implementadas com essa abordagem.

5.3.13 Implementação de variação *Modificar instrução SQL*

Em alguns pontos do código foi realizada a alteração de instruções SQL (*select*, *insert*, *update*, *delete*). Essas alterações foram classificadas como variação do tipo *Modificar instrução SQL*. Um exemplo de ocorrência dessa variação é a "Alt 128", *ticket* 9013, na qual foi realizada uma correção na consulta de produções do docente por área e sub-área. Essa alteração foi implementada com um `advice` do tipo `around`, e um *pointcut* que intercepta a execução do método `findByAcervo` da classe `ProducaoDao` no qual foi realizada a alteração. A implementação é apresentada na listagem 5.16. Também foi incluída nessa implementação, a "Alt 121", *ticket* 5854 que também é um exemplo de variação do tipo *Modificar instrução SQL*.

```
1       Collection<Producao> around(ProducaoDao dao, String titulo,
2                            Integer tipoProducao, Integer
                                areaConhecimento,
3                            Integer anoPublicacao, Integer
                                departamento)
4                            throws DAOException:
5       execution(public Collection<Producao> ProducaoDao.
           findByAcervo(..))
6       && target(dao)
7       && args( titulo,   tipoProducao,  areaConhecimento,
           anoPublicacao,
8             departamento)
9     {
10          //MUDANCA Ticket #5854 Problema ao listar acervos
               digitais
11          String hql = "select new Producao(p.id, p.titulo, p.
               servidor.id, p.servidor.pessoa.nome, "
12              + "p.tipoProducao.id, p.tipoProducao.descricao, p.
                   anoReferencia, p.servidor.unidade.sigla, p.
                   idArquivo)"
13              + " from Producao p "
14              + " join p.servidor s "
```

```
15              + "where p.idArquivo is not null";
16          String where = "";
17          if (titulo != null && !titulo.equals("")) {
18            titulo = "%"+ titulo;
19            String[] strings = titulo.split(" ");
20            StringBuilder resultado = new StringBuilder();
21            for (String string : strings) {
22              resultado.append(string + "%");
23            }
24            where += " and " + UFRNUtils.toAsciiUpperUTF8("p.titulo
                  ")
25                 + " like "
26                 + UFRNUtils.toAsciiUpperUTF8("'" + resultado.
                      toString() + "'");
27          }
28          if (tipoProducao != null) {
29            where += " and p.tipoProducao.id = " + tipoProducao;
30          }
31          if (areaConhecimento != null) {
32            //MUDANCA Ticket #9013 Correção da consulta por área e
                  sub-área
33            where += " and (p.area.id = " + areaConhecimento
34                +" or p.subArea.id = "+areaConhecimento+")";
35          }
36          if (anoPublicacao != null) {
37            where += " and p.anoReferencia = " + anoPublicacao;
38          }
39          if (departamento != null) {
40            where += " and p.servidor.unidade.id = " + departamento
                  ;
41          }
42          Query q = dao.getSession().createQuery(
43              hql + where + " order by  p.tipoProducao.id, p.
```

```
                    servidor.id, p.anoReferencia desc, p.titulo");
44          return q.list();
45      }
```

Listagem 5.16: Implementação da “Alt 128” e “Alt 121” com AspectJ

5.3.14 Implementação de variação *Modificar expressão condicional*

Alterações nas quais uma expressão condicional foi modificada foram classificadas em variação do tipo *Modificar expressão condicional*. Um exemplo dessa variação foi implementada na "Alt 84", *ticket* 20138, na qual uma expressão condicional foi modificada para adicionar uma nova condição 5.10b.

```
    @Override
    public void beforeCadastrarAfterValidate() throws NegocioException, SegurancaException,
DAOException {

        try // se é um discente cadastrado após a migração do antigo Prodocente
        {
            if (obj.getDiscenteExterno())


                obj.setOrientandoDiscente(null);

            obj.setDiscenteMigrado(false);
        } catch (Exception e) // se o discente foi migrado do antigo Prodocente:
        {
            obj.setDiscenteMigrado(true);
```

(a) Código original

```
  @Override
  public void beforeCadastrarAfterValidate() throws NegocioException, SegurancaException,
AOException {

      try // se é um discente cadastrado após a migração do antigo Prodocente
      {
          //MUDANCA #20138
          if (obj.getDiscenteExterno() || obj.getTipoOrientacao().getId() ==
ipoOrientacao.RESIDENCIA_MEDICA)
              obj.setOrientandoDiscente(null);

          obj.setDiscenteMigrado(false);
      } catch (Exception e) // se o discente foi migrado do antigo Prodocente:
      {
```

(b) Código modificado

Figura 5.10: Exemplo de alteração em expressão condicional

Na listagem 5.17 é apresentado um exemplo da implementação desse tipo de variação.

```
1       void around (TeseOrientadaMBean mBean) throws
            NegocioException, SegurancaException, DAOException:
2   execution(public void TeseOrientadaMBean.
        beforeCadastrarAfterValidate())
3   && target(mBean)     {
4   TeseOrientada obj = mBean.getObj();
5     try // se é um discente cadastrado após a migração do antigo
          Prodocente
6     {
7       //MUDANCA #20138
8       if (obj.getDiscenteExterno() || obj.getTipoOrientacao().
            getId() == TipoOrientacao.RESIDENCIA_MEDICA)
9         obj.setOrientandoDiscente(null);
10      obj.setDiscenteMigrado(false);
11    } catch (Exception e) // se o discente foi migrado do antigo
          Prodocente:
12    {
13      obj.setDiscenteMigrado(true);
14      obj.setOrientandoDiscente(null);
15    }
16    if ( !ValidatorUtil.isEmpty( obj.getPrograma()) &&
          ValidatorUtil.isEmpty( obj.getProgramaPos()) ){
17      obj.setProgramaPos(null);
18    }
19    mBean.superBeforeCadastrarAfterValidate();
20  }
```

Listagem 5.17: Implementação de variação do tipo *Modificar expressão condicional*

5.3.15 Implementação de variação *Excluir comando condicional*

A variação do tipo *Excluir comando condicional* foi introduzida em pontos do código nos quais uma expressão condicional foi removida após uma alteração. A figura 5.11 apresenta um exemplo desse tipo de variação. Nessa figura, é possível visualizar a "Alt 135", na qual foi realizada a exclusão de comando condicional do método `cancelar` da classe `TrabalhoFimCursoMBean`

```
@Override
public String cancelar() {
    resetBean();
    if (isPortalCoordenadorGraduacao())
        return forward("/graduacao/coordenador.jsf");
    else {
        if (preCadastroParaGraduacao || atualizacaoParaGraduacao) {
            String mbean = "registroAtividade";
            resetBean(mbean);
            return ((RegistroAtividadeMBean)getMBean(mbean)).cancelar();
        }
        return forward("/portais/docente/docente.jsf");
    }

}
```

(a) Código original

```
public String cancelar() {
    resetBean();
    //MUDANCA Ticket #8215 redirecionamento incorreto
    return forward(getListPage());
//      if (isPortalCoordenadorGraduacao())
//          return forward("/graduacao/coordenador.jsf");
//      else {
//          if (preCadastroParaGraduacao || atualizacaoParaGraduacao) {
//              String mbean = "registroAtividade";
//              resetBean(mbean);
//              return ((RegistroAtividadeMBean)getMBean(mbean)).cancelar();
//          }
//          return forward("/portais/docente/docente.jsf");
//      }
}
```

(b) Código modificado

Figura 5.11: Exemplo de exclusão de comando condicional

Para implementar esse tipo de variação, foi utilizado um *advice* do tipo `around` e um pointcut que intercepta o método `cancelar` e o código alterado é executado no lugar do

método interceptado. Na listagem 5.18 é apresentada a implementação da "Alt 135", extraída do Aspecto `CorrecaoRedirTrabalhoFimCursoMBean`.

```
1  String around(TrabalhoFimCursoMBean mBean):
2    execution(public String TrabalhoFimCursoMBean.cancelar())
3    && target(mBean)
4      {
5        mBean.resetBean();
6        return mBean.forward(mBean.getListPage());
7      }
```

Listagem 5.18: Exemplo de variação do tipo *Excluir comando condicional*

5.3.16 Implementação de variações *Modificar tipo de atributo* e *Modificar tipo de retorno do método*

Na figura 5.12 é apresentado um exemplo de alterações que foram classificadas como variações dos tipos *Modificar tipo de atributo* e *Modificar tipo de retorno do método*. Essas alterações foram realizadas na classe `AbstractControllerProdocente`.

Nas alterações “Alt 164” e “Alt 177” (*ticket* 11091), o tipo do atributo `mapaProducoes` da classe `AbstractControllerProdocente`. O tipo do atributo foi alterado de: *Map <Servidor, Collection<Producao> >* para *Map<Object, Collection <Producao> >*, assim como o retorno do método `getMapaProducoes`.

Para implementar essa alteração foi necessário realizar uma adaptação, pois não é possível modificar o tipo de retorno de um método ou o tipo de um atributo diretamente com AspectJ. Introduzir uma declaração *inter-type* com mesmo nome de atributo e método geraria um conflito com os atributos e métodos pré-existentes, também não foi possível utilizar um `around` em torno do método e retornar um tipo de dado diferente. Desse modo, a solução encontrada para implementar essas duas alterações foi declarar um novo atributo privado, restrito ao Aspecto, com um novo tipo e declarar novos métodos `get` e `set`.

```
protected Map<Servidor, Collection<Producao>> mapaProducoes;

public Map<Servidor, Collection<Producao>> getMapaProducoes() {

    return this.mapaProducoes;
}
```

(a) Código original

```
protected Map<Object, Collection<Producao>> mapaProducoes;

//MUDANCA Ticket #11091
public Map<Object, Collection<Producao>> getMapaProducoes() {
    return this.mapaProducoes;
}
```

(b) Código modificado

Figura 5.12: Comparação de alterações na classe `AbstractControllerProdocente`

O artifício descrito para adaptar este tipo de variação foi utilizado para implementar as alterações 165 e 177. Essa implementação é apresentada na listagem 5.19.

Foi declarado um novo atributo `mapaProducoes2` com um novo tipo. Após a declaração, foi realizada uma busca por referências ao atributo original `mapaProducoes` e trocadas essas referências pelo novo atributo `mapaProducoes2`, por isso o método `popularMapaProducoes` também foi modificado nesse Aspecto.

```
1  private Map<Object, Collection<Producao>>
       AbstractControllerProdocente.mapaProducoes2;
2  public Map<Object, Collection<Producao>> getMapaProducoes2() {
3    return this.mapaProducoes;
4  }
5  public void setMapaProducoes(Map<Object, Collection<Producao>>
       mapaProducoes2) {
6    this.mapaProducoes2 = mapaProducoes2;
7  }
8  String around(AbstractControllerProdocente acp, Collection<?
       extends Producao> producoes):
9    execution( protected void AbstractControllerProdocente.
10                popularMapaProducoes(..) )
11   && target (acp)
12   && args(producoes)
13   {
14   // Criar mapa de servidores e suas produções
15   //MUDANCA Ticket #11091
16   acp.mapaProducoes2 = new
17       TreeMap<Object, Collection<Producao>>(new Comparator<Object
             >() {
18   ...
19   acp.mapaProducoes2.put(docente, producoesServidor);
20     }
21     producoesServidor.add(p);
22   }
23 }
```

Listagem 5.19: Exemplo de implementação de variações dos tipos *Modificar tipo de atributo* e *Modificar tipo de retorno do método*

5.3.17 Implementação de variação *Modificar tipo genérico*

Nesse tipo de variação o tipo genérico <T> da classe foi modificado. A "Alt 111" é um exemplo deste tipo de variação encontrada no módulo de Produção Intelectual. Essa alteração foi realizada na classe `ImportTrabalhoEvento`. É possível observar na figura 5.13 que o tipo genérico da classe `ImportProducao` foi modificado de `PublicacaoEvento` para `Artigo`.

```
*/
public class ImportTrabalhoEvento extends ImportProducao<PublicacaoEvento> {
```

(a) Código original

```
*/
//MUDANCA Ticket #10932
public class ImportTrabalhoEvento extends ImportProducao<Artigo> {
```

(b) Código modificado

Figura 5.13: Comparação de alterações na classe `ImportTrabalhoEvento`

Como a classe `ImportTrabalhoEvento` já estende `ImportProducao` não é possível adicionar uma nova declaração *inter-type* do tipo `declare extends`. A solução adotada para implementar essa alteração foi: criar uma nova classe restrita ao Aspecto `ImportTrabalhoEventoArtigo` que estende `ImportProducao` com o tipo genérico modificado. Posteriormente, foi implementado um `advice` que intercepta a execução do método que utiliza a classe `ImportTrabalhoEvento` e substitui a utilização por `ImportTrabalhoEventoArtigo`. Na listagem 5.20, é apresentada parte da implementação.

```
1  void around(Curriculo cr, String uri, String localName, String
       qName, Attributes attributes) throws SAXException :
2    execution(public void Curriculo.startElement(String , String ,
       String , Attributes )    )
3    && args( uri,  localName,  qName,  attributes)
4    && target(cr)
```

```
5       {
6       ...
7         if ("PRODUCAO-BIBLIOGRAFICA".equals(qName)) {
8           ...
9           producao = new ImportTrabalhoEventoArtigo(input.
                getInputStream(), dao, producoes);
10          producao.read();
11          ...
12    }
13
14    private class ImportTrabalhoEventoArtigo extends ImportProducao<
          Artigo> {
15      public ImportTrabalhoEventoArtigo(InputStream input,
            ImportLattesDao dao, List producoes) {
16        super(input, dao, producoes, "TRABALHO-EM-EVENTOS");
17      }
18      @Override
19      public void startElement(String uri, String localName, String
            qName, Attributes attributes)
20          throws SAXException {
21          ....
22        }
23    }
```

Listagem 5.20: Exemplo de variação do tipo *Modificar tipo genérico*

5.4 Variação não implementada: *Modificar anotação*

Dentre os vinte e cinco tipos de variações introduzidas no módulo de produção intelectual, a única que não foi possível implementar foi a variação do tipo *Modificar anotação*. A "Alt 100", *ticket* 8412, foi classificada como variação do tipo *Modificar anotação*. Nessa alteração, os valores das anotações da classe `QualificacaoDocenteMBean` mo-

dificados. A anotação @Component("qualificacaoDocente") foi substituída por @Component("qualificacao") e a anotação @Scope("request") foi alterada para `@Scope("session")`, como pode ser observado na figura 5.14.

```
@Component("qualificacao")
@Scope("request")

public class QualificacaoDocenteMBean extends
        AbstractControllerAtividades<br.ufrn.sigaa.
```

(a) Código original

```
//MUDANCA Ticket #8412
@Component("qualificacaoDocente")
@Scope("session")
public class QualificacaoDocenteMBean extends
        AbstractControllerAtividades<br.ufrn.sigaa.
```

(b) Código modificado

Figura 5.14: Alteração realizada nas anotações da classe `QualificacaoDocenteMBean`

Na figura 5.14a, é possível visualizar o código original da UFRN e na figura 5.14b, o código modificado pela UFS.

Ao pesquisar a possibilidade de implementar essa alteração com a utilização da cons-

trução `declare @type`, foi observado que AspectJ não permite substituir o valor de uma anotação existente [7], isto é, permite apenas incluir uma anotação nova. Existe uma solicitação para incluir esse novo recurso no compilador em uma futura versão. A solicitação para essa modificação foi aberta em 2010, mas ainda não foi atendida [8]. Assim, devido a uma limitação de AspectJ, não foi possível implementar a "Alt 100".

5.5 Teste das alterações

Após a codificação dos Aspectos, foi iniciada a terceira etapa do estudo de caso, que compreende o processo de testes da implementação das variações introduzidas no módulo de Produção Intelectual, dentro do ambiente de desenvolvimento da UFS. O processo de teste se resume a executar o programa com a finalidade de encontrar erros, defeitos ou falhas, por isso testes são feitos para adicionar valor a um programa, aumentando sua qualidade e confiança (MYERS; SANDLER; BADGETT, 2011). Após a realização dos testes demonstra-se que o *software* aparentemente está em conformidade com a sua especificação e planejamento, de forma que os requisitos tenham sido atendidos (PRESSMAN, 2011). Os testes dos Aspectos implementados foram realizados no NTI/UFS.

Devido ao constante número de alterações realizadas no SIG, a equipe de desenvolvimento da UFS adotou um processo de teste de software parcialmente automatizado para verificar a integridade do sistema, após as alterações introduzidas pela UFS. Antes de liberar uma nova versão do sistema para o ambiente de produção, é realizada uma série de testes automatizados baseados nas especificações do sistema (SANTOS, 2015). Caso sejam registradas falhas na execução, os problemas são registrados no *Redmine* para correção dos defeitos encontrados. Para execução dos testes automatizados, são utilizadas as ferramentas *Selenium IDE*, Eclipse e JUnit [9]/Webdriver. O *Selenium IDE* é um ambiente para gravação e reprodução (*Record/Playback*) das ações do usuário. Dentre os artefatos gerados após a execução dos testes, o Selenium possibilita que os casos de teste sejam armazenados em arquivos *HyperText Markup Language* (HTML) e em *scripts* para serem utilizados no Eclipse

[7] `http://stackoverflow.com/questions/4106187`
[8] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=313026`
[9] `http://junit.org/`

com as bibliotecas do *JUnit* e *Webdriver*. A biblioteca do *Webdriver* possibilita que os testes automatizados sejam executados em diversos navegadores web, incluindo Chrome, Firefox e Internet Explorer. Um caso de teste consiste em um conjunto de condições estabelecidas para possibilitar o teste de software (MYERS; SANDLER; BADGETT, 2011). A utilização do *Selenium* possibilita à equipe de testes da UFS gravar e salvar a execução de casos de teste e repetir as verificações sempre que uma nova versão do SIGAA é disponibilizada para homologação e produção.

Foi disponibilizado, pela Coordenação de Sistemas (COSIS) do Núcleo de Tecnologia de Informação (NTI/UFS), um computador com a seguinte especificação: processador Intel® Core™ i5, com 4 Gb de memória RAM e Sistema Operacional Windows 7 (64 bits). Os computadores do NTI, utilizados no setor de desenvolvimento de sistemas possuem essa especificação padrão. No computador com essa especificação, foi instalada a versão 2.2.3 do AJDT (e versão 1.7.3 do AspectJ). Nesse computador, com o ambiente de desenvolvimento padrão, adotado pela equipe de desenvolvimento da UFS, foram utilizados:

- Eclipse IDE *for Java EE Developers* versão 4.4.
- Servidor de aplicação JBoss 4.2.
- Selenium IDE [10] 1.0.4.
- Mozilla Firefox 37.0.2.

Os testes foram conduzidos na base de dados de desenvolvimento do SIGAA (versão 3.5.15), utilizada pela coordenação de sistemas para a realização de testes. Essa base é atualizada pela equipe de banco de dados e mantida em consistência com a versão atual do SIGAA.

Para verificação das alterações introduzidas no módulo de produção intelectual com AspectJ, foi adotado o seguinte processo de teste, com base no processo de teste da UFS (SANTOS, 2015):

1. Elaboração dos casos de teste.

[10] www.seleniumhq.org

2. Execução do teste com o *Selenium*.

3. Observação dos resultados.

4. Realização de correções no código, caso alguma falha fosse detectada.

5. Captura dos resultados, coleta e armazenamento dos artefatos.

Nas próximas subseções apresenta-se o processo adotado, utilizando como exemplo um caso de teste da "Alt 77", *ticket 8225*. Nessa alteração, foi adicionado um comando condicional para realizar a validação entre a data da defesa e data de início do período de orientação no cadastro de uma tese orientada.

5.5.1 Casos de teste

Com base nas descrições dos *tickets* apresentados no *Redmine* foram elaborados os casos de teste. Em um documento, foram inseridas a descrição de todas as alterações (requisitos) a serem testados, incluindo as seguintes informações:

ID do caso de teste: CT 8225

Pré-condições: O sistema deve estar disponível para acesso pela URL: O sistema deve ser acessado por um usuário docente, que tenha acesso ao módulo de Produção Intelectual.

Objetivo do caso de teste: Verificar a validação de datas em Tese orientada.

Passo a Passo:

1. Acessar o “SIGAA”, o “Portal do Docente”, no menu “Outras Atividades” -> “Orientações” -> “Orientações Pós-Graduação”;

2. Clicar na funcionalidade “Cadastrar Nova Orientação”;

3. Preencher todos os campos obrigatórios com dados válidos e no campo “Data de Defesa” digitar uma data menor do que a data de início do período de orientação;

4. Clicar no botão “Cadastrar”.

Especificações de entrada: Os campos obrigatórios devem ser preenchidos com dados válidos e no campo "**Data de Defesa**" deve ser inserida uma data menor do que a data de início do período de orientação.

Especificações de saída (Resultado esperado): O sistema deveria exibir uma mensagem informando que a data da defesa é menor do que a data de início do período de orientação.

Nessa alteração, por exemplo, o sistema não estava realizando a validação entre a data da defesa e data de início do período de orientação. Antes de ser adicionada a modificação, o registro era adicionado, mesmo com a informação incorreta, apresentando a mensagem de confirmação do cadastro: "*Operação realizada com sucesso!*". Após a alteração, deseja-se verificar nesse caso de teste, se a validação está sendo realizada como deveria.

5.5.2 Execução dos testes com o *Selenium* e observação dos resultados

Utilizando o Selenium IDE e o navegador *Mozilla Firefox* foram executados os casos de testes necessários para verificar as alterações implementadas e possibilitar a detecção de falhas na implementação.

O procedimento adotado para executar os casos de teste foi o mesmo:

1. Iniciar o sistema pela URL base *http://localhost:8080/sigaa/*,
2. Acessar a página inicial */sigaa/public/home.jsf*
3. Entrar no sistema
4. Executar o passo a passo previsto nos Casos de Teste.

Na figura 5.15, é apresentado o ambiente *Selenium*, utilizado com o Mozilla para realização dos testes. Com a função gravar (*Click to record*) do *Selenium* ativada, foi possível salvar os passos de cada um dos casos de testes. Assim, foi possível refazer novamente os casos de teste quando houve a detecção de alguma falha na execução, após a correção da alteração.

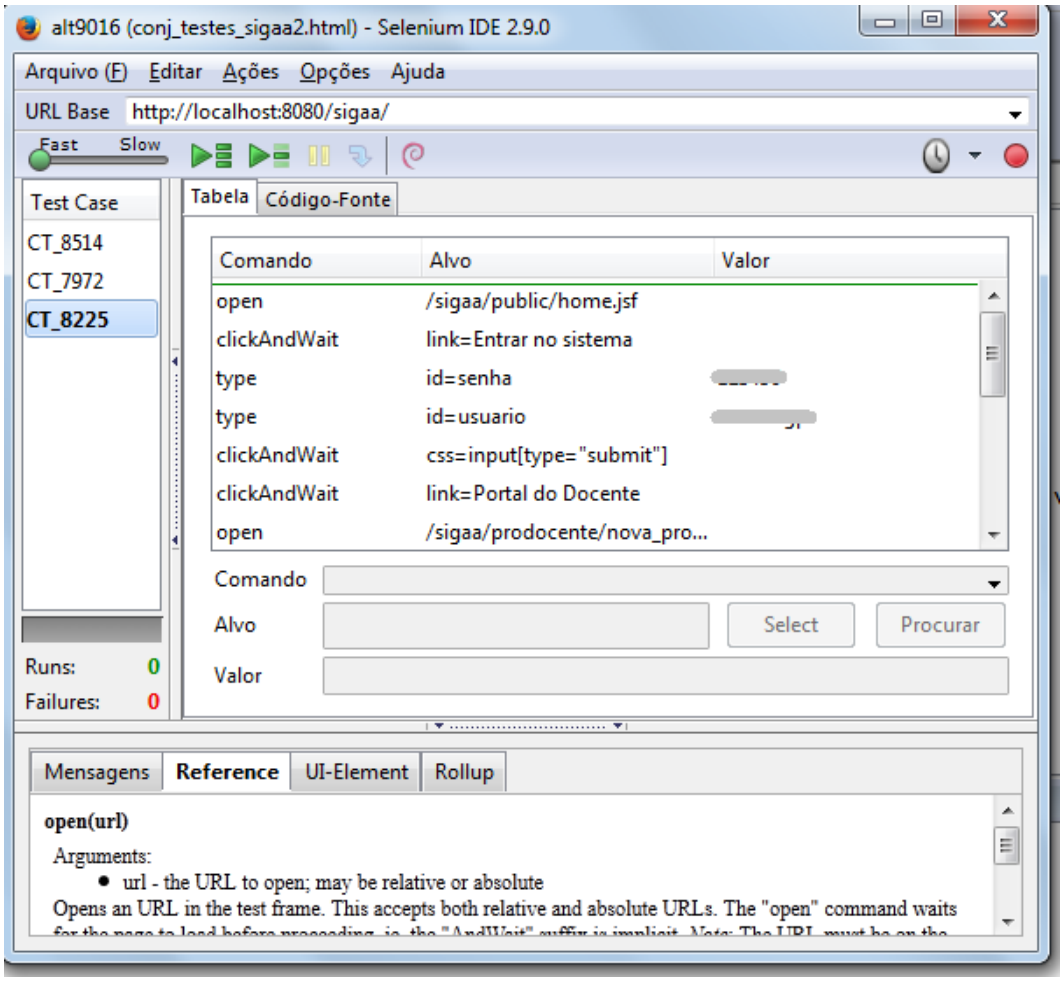


Figura 5.15: *Selenium IDE* utilizado para gravação e armazenamento dos testes realizados

Na figura 5.16, é apresentado o resultado da execução do caso de teste **CT 8225** e a gravação com o *Selenium*. É possível observar que o resultado esperado no caso de teste foi alcançado com sucesso.

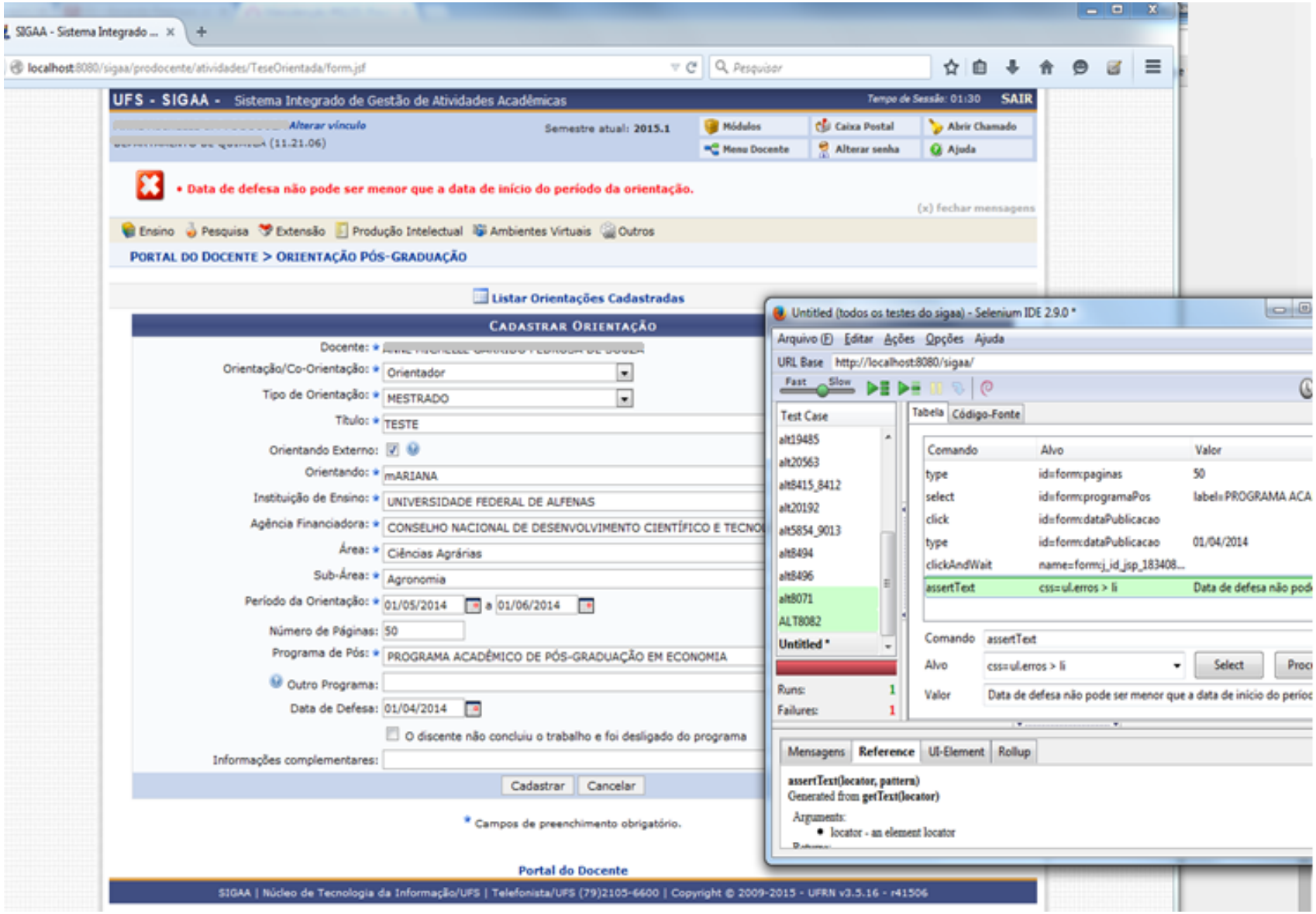


Figura 5.16: Resultado da execução do caso de teste **CT 8225**

Durante esse processo, ao detectar alguma falha na implementação, a correção do código foi realizada e o teste executado novamente, até que todos os Aspectos implementados fossem testados.

5.6 Atualização de versão do módulo de Produção Intelectual

Nessa seção, é apresentada a etapa desse trabalho que compreendeu a atualização do módulo de Produção Intelectual do SIGAA, utilizando os Aspectos previamente apresentados.

Foi entregue pela COSIS, a versão 3.15.24 do SIGAA, a última versão do sistema fornecida pela UFRN. Essa versão do SIGAA foi comparada com a versão do SIGAA utilizada na UFS (3.5.15). Foi observado que o módulo de Produção Intelectual passou por modificações, foram adicionados nove (9) pacotes e cento e vinte e oito classes (128). Uma comparação realizada com o *WinMerge* demonstrou a existência de cento e sessenta e sete (167) conflitos nos arquivos do módulo.

De posse da nova versão do SIGAA, após inserir as bibliotecas necessárias, foram adicionados os Aspectos implementados. Foi realizada a compilação do projeto e após o *weaving* dos Aspectos foram observados os erros informados pelo Eclipse, com a finalidade de avaliar os resultados da atualização.

Conforme os Aspectos foram adicionados à nova versão do SIGAA, observou-se que algumas alterações implementadas nos Aspectos apresentaram erros devido à falta de alterações realizadas pela UFS em outros módulos, como ensino e pesquisa. Como essas alterações, realizadas fora do módulo de Produção Intelectual não foram implementadas nesse estudo de caso, alguns elementos foram comprometidos na atualização. Esse problema foi observado nos seguintes pontos:

$Erro_1$**:** No Aspecto `IncluirAlunosAtivos`, foi implementada a "Alt 19", nessa alteração foi modificado o método `getXmlContent` da classe `DiscenteServlet` para adicionar um comando condicional. Ao incluir essa alteração, foi utilizada a constante `GRADUANDO` da classe `StatusDiscente`. Essa constante define o discente de graduação

que integralizou todo currículo do curso e aguarda a colação de grau. A classe `StatusDiscente` não faz parte do sistema SIGAA, mas sim da biblioteca de classes "comum.jar", do projeto "02_EntidadesComuns", que é utilizada em todos os sistemas do SIG. A versão 3.15.24 do SIGAA utiliza a versão 1.4.2 da biblioteca *comum*. Nessa versão, mais atual que a versão utilizada pela UFS no SIGAA (versão 3.5.15), a constante `GRADUANDO` foi excluída da classe `StatusDiscente`.

Para não ter que realizar o `weaving` da biblioteca de classes `comum`, foi inserida a constante `GRADUANDO` localmente no Aspecto `IncluirAlunosAtivos`.

$Erro_2$: Nessa mesma alteração ainda há outro problema, o valor da constante `GRADUANDO` na versão utilizada pela UFS é igual a 9, que na versão mais recente da classe, é o valor de outra constante, a constante `FORMADO`. Na versão anterior da classe `StatusDiscente`, utilizada atualmente pela UFS, essa constante `FORMADO` não existe. Diante desse problema, torna-se evidente a necessidade de alterar o valor da constante `FORMADO` para outro valor inexistente, já que possivelmente a atualização irá gerar um conflito de informações na base de dados atual da UFS, em que alunos com o status de `GRADUANDO` seriam confundidos com o status de `FORMADO`. Com isso, é possível observar que, a atualização dessa alteração, gerou a necessidade de uma nova alteração na classe `StatusDiscente`, nos locais em que a constante `FORMADO` é utilizada.

$Erro_3$**:** No Aspecto `GeracaoIPICalculoAreaProjetoDocente`, foi adicionado o método `isAreaProjetoDocente` à classe `CalculosPesquisaImplUFRN`. Nesse método, são utilizadas as constantes `IC` e `IT` que pertencem a classe `CategoriaProjetoPesquisa`. Contudo, essas constantes foram inseridas pela UFS em alterações referentes (`tickets` 10336, 12121) ao módulo de Pesquisa do SIGAA e não foram encontradas na versão mais recente do sistema atualizado. Por isso, houve a necessidade de inserir essas duas constantes, que foram adicionadas com uma declaração `inter-type`, no Aspecto `GeracaoIPICalculoAreaProjetoDocente`.

Na listagem 5.21, é apresentado o código do método `isAreaProjetoDocente` e as constantes adicionadas, que fazem parte do Aspecto `GeracaoIPICalculoAreaProjetoDocente`.

```
1   public aspect GeracaoIPICalculoAreaProjetoDocente {
2   public static final int CategoriaProjetoPesquisa.IC = 1;
3   public static final int CategoriaProjetoPesquisa.IT = 2;
4   public boolean CalculosPesquisaImplUFRN.isAreaProjetoDocente(
5   AreaConhecimentoCnpq area, int idDocente,
6   int idRelatorio, Integer anoVigencia) throws DAOException{
7       ProjetoPesquisaDao projetoDao = DAOFactory.getInstance().
            getDAO(ProjetoPesquisaDao.class);
8       EditalPesquisa edital = new EditalPesquisa();
9       List<ProjetoPesquisa> projetos = new ArrayList<
            ProjetoPesquisa>();
10      //MUDANCA #20998 preenche o edital antes de pesquisar
11      if (idRelatorio == RelatorioProdutividade.
            RELATORIO_DISTRIBUICAO_COTAS_PIBIC_POR_AREA){
12          edital = projetoDao.findByExactField(
13          EditalPesquisa.class, new String[]{"categoria.id", "
                edital.ano"},
14          new Object[]{(Integer)CategoriaProjetoPesquisa.IC,
                anoVigencia}, true);
15      } else if (idRelatorio == RelatorioProdutividade.
            RELATORIO_DISTRIBUICAO_COTAS_PIBIT_POR_AREA){
16          edital = projetoDao.findByExactField(
17          EditalPesquisa.class, new String[]{"categoria.id", "
                edital.ano"},
18          new Object[]{(Integer)CategoriaProjetoPesquisa.IT,
                anoVigencia}, true);
19      }
20      ...
21      return false;
22    }
```

Listagem 5.21: Alterações do módulo de pesquisa inseridas no aspecto `GeracaoIPICalculoAreaProjetoDocente` após a atualização

$Erro_4$: No Aspecto `GeracaoIPICalculoAreaProjetoDocente`, a implementação da alteração (*ticket* 19480) apresentou um erro pois as classes `Classificacao-Relatorio` e `EmissaoRelatorio` foram modificadas pela UFS em alterações relacionadas ao módulo de pesquisa (*tickets* 11341 e 11454). Essas alterações não foram implementadas nos Aspectos, pois não faziam parte do módulo de produção intelectual. Com isso, após a atualização houve a necessidade de inserir também essas alterações. Nesse Aspecto, foram inseridos o campo `editaisSelecionados` na classe `ClassificacaoRelatorio` (e os métodos `get` e `set`) e na classe `EmissaoRelatorio`, foi adicionado o campo `docenteExterno` (e os métodos `get` e `set`).

$Erro_5$**:** Outro problema aconteceu com o método `findByDocenteEdital` da classe `ProjetoPesquisaDao` que também era referenciado no Aspecto `GeracaoIPICalculo-AreaProjetoDocente`. Como não havia nenhuma marcação com a *tag* MUDANCA nesse método, foi verificada a versão anterior às modificações da UFS e observou-se que esse método foi inserido pela UFS, porém nenhuma marcação foi realizada. Assim, para corrigir o erro nesse Aspecto, o método `findByDocenteEdital` foi inserido com uma declaração *inter-type*. Essas alterações resolveram os problemas de compilação no Aspecto `GeracaoIPICalculoAreaProjetoDocente`.

$Erro_6$**:** No `Aspecto VerificaoFormacaoDuplicada`, foi implementada a "Alt 125", `ticket 15156`, na qual foi inserido um `pointcut` que interceptava a execução do método `findFormacaoAcademicaDoServidor` da classe `FormacaoAcademicaDao` e foi utilizado um *advice* do tipo `around` para executar o código modificado. O método retornava como resultado uma coleção de dados do tipo `FormacaoAcademica`: `Collection<FormacaoAcademica>`. Porém, após a atualização, o Aspecto deixou de funcionar, uma vez que as classes `FormacaoAcademicaDao` e `FormacaoAcademica` foram excluídas na versão mais recente do módulo. Diante desse erro, foi registrado um caso de `pointcut` que após a atualização deixou de funcionar corretamente.

Após a atualização, foi realizada uma inspeção manual dos Aspectos implementados, isto é, cada Aspecto foi inspecionado com o objetivo de detectar problemas em *pointcuts* que poderiam ter deixado de funcionar ou que estariam capturando pontos incorretos após a atualização. Contudo, além dos problemas apresentados, não foram detectados outros erros em *pointcuts* implementados. Em seguida à realização dessa etapa, a implantação do sistema deveria ter sido realizada, para verificação em tempo de execução do comportamento dos Aspectos, pelos casos de teste previamente definidos. Porém, para a implantação, seria necessário realizar as atualizações nos esquemas de bancos de dados e na base de dados, utilizada atualmente no sistema em produção pela UFS. No entanto, há um distanciamento muito grande entre as versões do SIGAA atualmente utilizada pela UFS (3.5.15) e a versão atual da UFRN (3.15.24). Observou-se que o número de atualizações foi muito alto, somente no módulo de produção intelectual, foi detectada a adição de cento e vinte e oito (128) classes e nove (9) pacotes.

Para implantar a versão atual do SIGAA na UFS, seria necessário o auxílio e a cooperação da UFRN bem como da UFS, o que atualmente não foi possível, pois os servidores da UFS e demais instituições federais de ensino superior paralisaram suas atividades desde o dia 28 de maio de 2015. Além disso, a COSIS está operando com um *déficit* no número de desenvolvedores, pois o contrato de funcionários terceirizados estava suspenso na época da realização desse estudo de caso. Assim, dentro do tempo previsto para realização dessa etapa do trabalho, não seria interessante para a UFS interromper o atendimento às demandas internas, e auxiliar na implantação do sistema, já que essa atividade não é uma prioridade da coordenação.

Diante da impossibilidade de realizar o teste das atualizações, com o sistema implantado, foi realizada somente a inspeção manual dos Aspectos após a atualização e obtidos os resultados descritos nessa seção.

Capítulo 6

Avaliação dos resultados

Nesse capítulo são apresentados os resultados obtidos a partir da análise das métricas e as respostas às questões de pesquisa.

6.1 Problemas encontrados durante o levantamento de variações

Durante o levantamento, foram observados alguns problemas no processo adotado atualmente pela equipe de desenvolvimento da UFS, para introdução das alterações. Foram identificados os seguintes problemas:

***Ticket* de identificação da tarefa não foi encontrado:** Em alguns locais, a identificação do *ticket* associada à *tag* **MUDANCA** não foi inserida. Com isso não é possível identificar qual tarefa está associada àquela alteração introduzida.

Um exemplo de alteração que foi introduzida sem o *ticket*, é apresentado na figura 6.1. Para identificar essa alteração, foi preciso encontrar qual tarefa no *Redmine* estava associada à classe alterada. Contudo, caso a classe tenha sido alterada diversas vezes, se torna impossível identificar qual tarefa está associada à alteração.

Na figura 6.1, é apresentado um trecho de código da classe `MiniCursoMBean`, em que a alteração de *ticket* 5797 não foi identificada. Foi necessário realizar uma leitura

```
//MUDANCA: adicionando validação das datas
@Override
public String cadastrar() throws SegurancaException, ArqException, NegocioException {
    ListaMensagens erros = new ListaMensagens();
    if(obj.getDataFim() != null && obj.getPeriodoInicio() !=null){
        ValidatorUtil.validaOrdemTemporalDatas(obj.getPeriodoInicio(), obj.getDataFim(), true

                "Período da Ação", erros);
    }

    if(!erros.isEmpty()){
        addMensagens(erros);
        return forward(getFormPage());
    }else{
        return super.cadastrar();
    }
}
```

Figura 6.1: Alteração realizada sem ticket na classe `MiniCursoMBean`.

do código e da tarefa para identificar a alteração. Esse problema se repete nos *tickets* 5805, 5816, 5817, 5850 e 5851 que não foram identificados, nas alterações introduzidas respectivamente nas classes `ExposicaoApresentacaoMBean`, `MontagemMBean`, `ProgramacaoVisualMBean`, `ParticipacaoSociedadeMBean` e `ParticipacaoColegiadoComissaoMBean`. Possivelmente a falta do *ticket* foi causada por esquecimento durante a implementação dessas alterações. Foi observado também que a classe `AvaliacaoDocenteDao` possui muitas alterações que não foram identificadas com os *tickets* de mudança.

Alterações sobrepostas: Observou-se em algumas classes, que as alterações no código podem se sobrepor, o que impossibilita a identificação visual de em qual parte do trecho de código se encontra cada alteração, apenas utilizando a *tag* **MUDANCA**. Na figura 6.2, é apresentado um trecho de código da classe `RelatorioProdutividadeMBean`, uma das quais em que esse tipo de problema foi identificado.

```
      //MUDANCA #19197 Alterar a geração do relátorio da avaliação para Concessão de Cota
      if (anoVigencia < 2014){
          //MUDANCA Ticket #11087
          //MUDANCA Ticket #11306 Adicionando busca por docente externo
          obj = RelatorioHelper.montarRelatorioProdutividade(
                  dao.findByPrimaryKey( idRelatorio, RelatorioProdutividade.class),
                  (isAcessoPrivilegiado()
                          ? ((docenteRelatorio.getId() != 0) ? docenteRelatorio :
docenteExternoRelatorio)
                          : docente),
                  anoVigencia, serviceFormacao);
      } else {
          ProjetoPesquisaDao projetoDao = getDAO(ProjetoPesquisaDao.class);
          EditalPesquisa edital = null;
```

Figura 6.2: Exemplo de alterações sobrepostas na classe `RelatorioProdutividadeMBean`

Erro de escrita: A *tag* MUDANCA foi escrita de forma incorreta (MUNDACA), o que impossibilita a busca textual pelo termo para a identificação de alterações. Na figura 6.3, é apresentado um trecho de código da classe `FormacaoAcademicaDao` na qual ocorre esse problema.

```
        //MUNDACA #15156
        String sql = "SELECT * FROM pessoal.formacao_academica f WHERE f.id_servidor =
idServidor +" AND f.ativo = true";
```

Figura 6.3: Erro de escrita na *tag* MUDANCA

Alterações *crosscutting*: Algumas alterações idênticas que se repetem em vários pontos do código fonte da aplicação. Essas alterações são consideradas *crosscutting*, por estarem espalhadas e entrelaçadas por vários trechos de código. Na figura 6.4, é apresentado um trecho de código da classe `ImportTrabalhoEvento` em que a alteração com *ticket* 20223 foi inserida. Essa alteração também foi realizada em outras 2 classes :`ImportArtigoPublicado` e `ImportJornalRevista`. Foi observado que este tipo de alteração ocorre outras vezes em outros pontos do código, nos quais também foi encontrado código duplicado.

```
//MUDANCA #20223 corrigindo erro na importacao lattes
item.setClassificacaoQualis(null);
```

Figura 6.4: Exemplo de alteração *crosscutting*

Alterações espalhadas: Para a realização de algumas mudanças no código, observou-se que, em alguns casos, foram introduzidas muitas modificações em vários pontos espalhados pelo módulo. A implementação do requisito para possibilitar a utilização por docente externo à instituição no módulo de Produção Intelectual (*ticket* **11091**) resultou em oitenta e sete (87) alterações no código. O que implicou na necessidade de alterar oitenta e sete pontos do código-fonte para introdução dessa nova funcionalidade. A ocorrência de alterações espalhadas como essa é mais um indício que reforça a utilização da POA para modularização desse tipo de requisito.

Identificação de maus cheiros *bad smells*: Durante a leitura e análise das variações introduzidas, foi observada a existência de alguns maus cheiros no código que poderiam ter sido melhorados por meio de uma refatoração. Refatorações são alterações feitas na estrutura interna do *software* para torná-lo mais fácil de entender e mais barato de modificar sem

alterar o seu o comportamento, eliminando a duplicação de código e tornando o *design* e a lógica mais claros (FOWLER, 1999). Ao analisar um código é possível encontrar estruturas de códigos nas quais se identifica a necessidade de refatoração, maus cheiros podem ser definidos como a sensação de que algo está errado com esse código (FOWLER, 1999). Dentre os problemas observados nas alterações realizadas, foi possível identificar alguns maus cheiros que indicam a necessidade de refatoração do código. Foram identificados os seguintes problemas:

- **Código duplicado:** O mesmo código foi inserido em classes ou métodos diferentes para introduzir uma alteração, como exemplificado na figura 6.4.
- **Métodos longos:** Métodos que foram alterados com a introdução de funcionalidades e tornaram-se longos demais. Exemplo: o método `montarRelatorio` da classe `RelatorioProdutividadeMBean`, que possui 136 linhas de código.
- **Aumento de complexidade do código:** Foram observadas algumas classes com muitos métodos e um número muito grande de linhas de código. As classes apresentadas no gráfico 6.5 possuem um número alto de linhas de código e esse número aumentou ainda mais com a introdução das alterações realizadas. Observa-se que foram acrescentadas quase 200 linhas de código à classe `RelatorioProdutividadeMBean` e 600 linhas à classe `AvaliacaoDocenteDao`.
- **Alteração divergente:** Uma mesma classe que sofreu alterações várias vezes, de diversas maneiras e por motivos diferentes. O número de alterações apresentado no gráfico da figura 6.5 corresponde ao número de *tags* MUDANCA inseridas no código pela equipe da UFS. É possível observar que um número alto de mudanças foi realizado nas classes apresentadas.
- **Cirurgia com rifle:** uma mesma alteração que implica em mudanças em vários locais diferentes do código. Exemplo: a alteração do *ticket* 7790 implicou em nove (9) mudanças em classes e a do *ticket* 11091 implicou na introdução de oitenta e sete (87) mudanças em pontos diferentes do módulo.

Uma análise do código-fonte de todo o sistema possivelmente indicaria a existência de

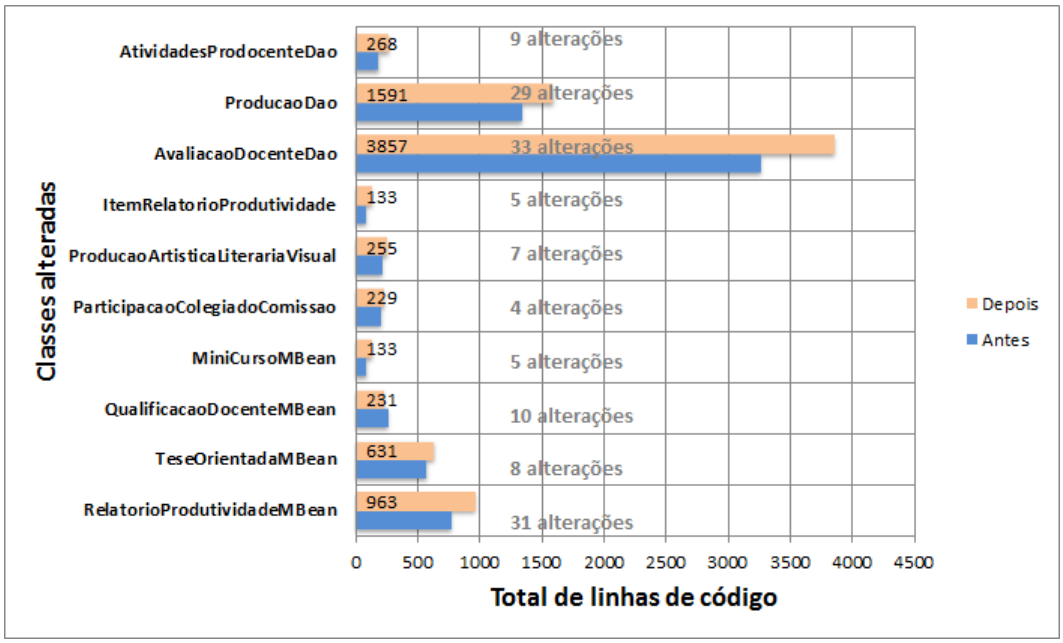


Figura 6.5: Quantidades de linhas de código de algumas classes do módulo de Produção Intelectual, antes e depois das alterações

outros maus cheiros e problemas, contudo esse não é o objetivo principal deste trabalho. A refatoração do código do SIG não é interessante para a UFS, pois não se deseja alterar o código base além do necessário para inserir os novos requisitos; por isso o código não foi analisado com o objetivo de identificar outros maus cheiros. Contudo, percebe-se que a introdução das alterações agravou ainda mais alguns problemas, como os que foram citados nesta seção.

6.2 Análise de implementação das variações

Esta seção apresenta uma avaliação com base na implementação das variações introduzidas pela UFS no módulo de Produção Intelectual. Nessa etapa, foram avaliadas a quantidade de tipos de variações catalogados, implementadas e não implementadas e a quantidade de alterações catalogadas, implementadas e não implementadas.

- Quantidade de tipos de variações catalogadas. Foram catalogados vinte e quatro (24) tipos de variações. Detalhados na tabela 5.1

- Quantidade de variações encontradas. Foram encontradas trezentas e onze (311) vari-

ações.

- Quantidade de tipos de variações implementados com AspectJ. Foram implementados vinte e três (23) tipos de variações.
- Quantidade de variações implementadas com AspectJ. Foram implementadas trezentas e dez (310) variações.
- Quantidade de variações não implementados com AspectJ. Não foi possível implementar uma variação do tipo *Modificar anotação*, "Alt 100", *ticket* 8412.

Conforme apresentado na seção 5.4, não foi possível implementar o tipo de variação *Modificar anotação* devido a uma limitação do AspectJ. Apesar de ser uma alteração considerada simples, o AspectJ não permite que uma anotação existente seja modificada. Com isso, a alteração 100 não foi implementada. Uma possibilidade para implementar essa alteração, por conta da limitação encontrada, sem modificar o código original da classe `QualificacaoDocenteMBean`, seria criar uma nova classe com a anotação alterada. Porém, seria necessário modificar outros pontos do código para alterar as referências à classe antiga, de modo que o custo dessa abordagem pode superar os benefícios.

Em algumas alterações, a implementação com AspectJ não foi tão simples e direta, por conta da necessidade imposta pela alteração e das limitações impostas pela linguagem. Nestes casos, buscou-se uma alternativa à implementação da alteração sem afetar o funcionamento da mesma.

Também houve casos de tipos de variações que, devido à falta de recursos da linguagem, foram implementados parcialmente como no caso das variações do tipo *Excluir método* e *Excluir atributo*.

No gráfico da figura 6.6, é apresentado o percentual da quantidade de alterações implementadas, implementadas parcialmente e implementadas com adaptações. É possível observar que o percentual de alterações implementadas diante do total de alterações encontradas demonstra que a utilização de AspectJ possibilitou a implementação de 99,68% das variações.

Os tipos de variações implementados com adaptações foram: *Adicionar chamada a mé-*

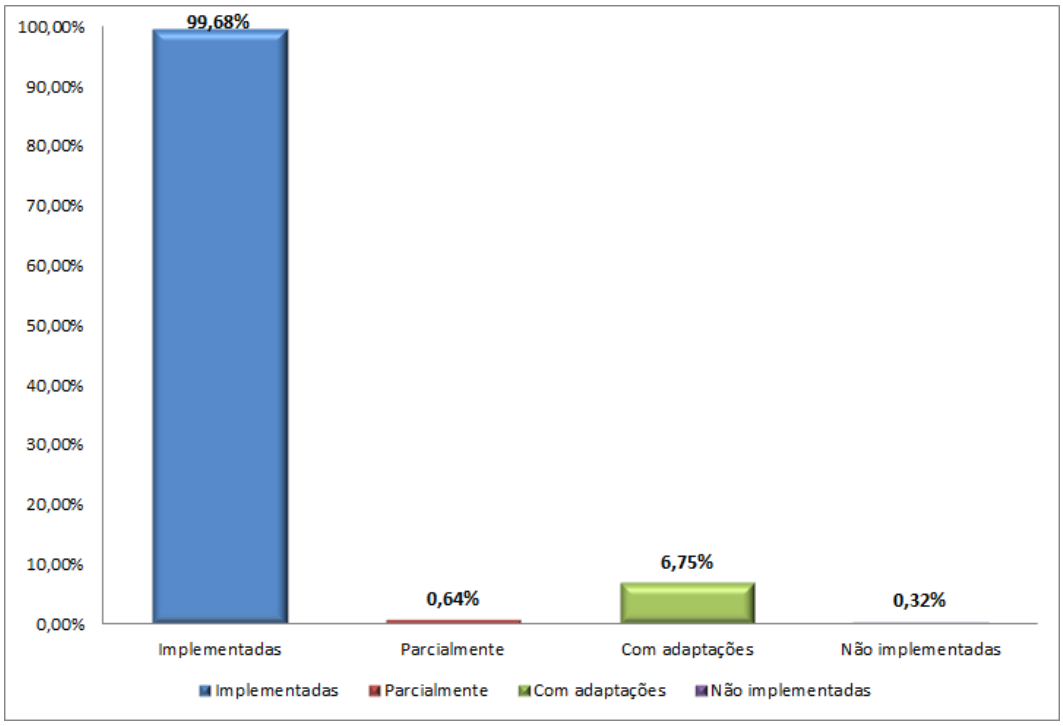


Figura 6.6: Percentual de variações "implementadas", "parcialmente implementadas", "implementadas com adaptações"e "não implementadas"com AspectJ

todo e *Adicionar método*, quando foram realizadas chamadas a métodos de superclasse, e *Adicionar novo construtor*. Os tipos de variações implementados parcialmente foram *Excluir método* e *Excluir atributo*.

Diante dos resultados obtidos após a implementação das variações com POA, observa-se que em relação à quantidade de variações implementadas no estudo de caso, a quantidade de variações implementadas com AspectJ é menor que a quantidade de variações catalogadas.

6.2.1 Impressões e problemas encontrados

Para o desenvolvimento e teste das implementações, foi utilizado um computador fornecido pela UFS, com as especificações apresentadas na seção 5.5. Nesse computador, foi observado um aumento no tempo de "compilação" ao realizar o *build* do projeto inteiro (SIGAA) com todos os Aspectos implementados.

O mesmo problema foi detectado durante a compilação do projeto também em outros dois computadores. Também foi observado um aumento no consumo de memória pelo

Eclipse após realizar o *weaving* do projeto que resultava em um erro de memória do compilador, conforme apresentado na figura 6.7. Porém, como esta análise do compilador de AspectJ não faz parte do escopo deste trabalho, não foi conduzida uma análise aprofundada deste problema.

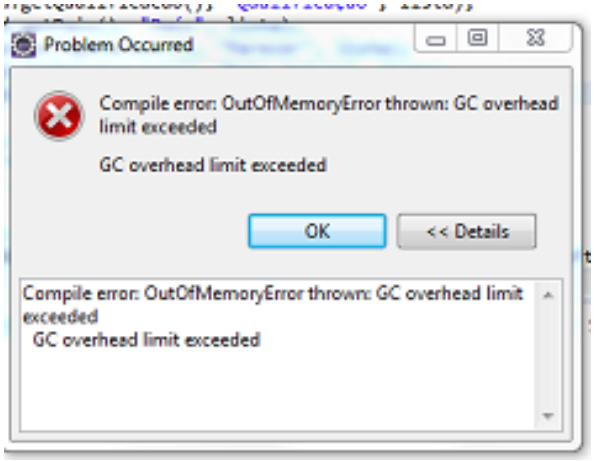


Figura 6.7: Erro de memória do compilador

Foi observado também que em alguns casos, como no exemplo da implementação da variação *Modificar valor de atributo*, apresentada na seção 5.3.7, apesar de a variação introduzida não ser tão complexa, há um esforço maior para implementação em Aspectos em comparação a modificação realizada pela UFS. Tanto em relação à necessidade de análise e adaptação quanto em relação à quantidade de código escrito para implementar a variação com AspectJ. Esse esforço e as adaptações são necessários por particularidades e limitações da própria linguagem e paradigma, como discutido na seção 5.3.

6.3 Avaliação da atualização

Conforme a descrição das atividades conduzidas nessa etapa e os resultados obtidos, descritos na seção 5.6, para a métrica da **quantidade de erros solucionados** após a atualização, foi observado que a hipótese alternativa (H_2) foi satisfeita e que (H_1) foi rejeitada. Esse resultado indica que o número de erros solucionados foi menor que o número de erros detectados. Foram detectados seis erros durante a atualização, esses erros são detalhados na seção 5.6. Dentre esses, foram corrigidos apenas $Erro_1$, $Erro_3$, $Erro_4$ e $Erro_5$. O $Erro_2$ precisa

de uma resolução de conflito devido à evolução da classe `StatusDiscente` e o $Erro_6$ apresenta um problema grave, pois duas classes foram removidas da versão mais recente do módulo `FormacaoAcademicaDao` e `FormacaoAcademica`.

6.4 Avaliação dos desenvolvedores

Com a finalidade de apresentar a equipe de responsáveis pela manutenção do SIG, os resultados obtidos com a realização deste estudo de caso, foi realizada uma apresentação de 25 minutos na qual foram abordados os seguintes tópicos:

- Dificuldades encontradas na utilização da abordagem atual pela UFS.
- Requisitos transversais.
- Programação Orientada a Aspectos.
- AspectJ.
- Como introduzir mudanças no SIG utilizando AspectJ.
- Resultados obtidos na implementação de variações do módulo de produção intelectual com AspectJ.
- Resultados obtidos na atualização de versão do módulo de produção intelectual com AspectJ.
- Benefícios e obstáculos observados.

Após a realização da apresentação, foi entregue o questionário do **Apêndice A**, contendo 12 perguntas, para os desenvolvedores presentes na apresentação. Foram preenchidos 12 questionários, contudo um destes não foi considerado durante a análise de dados, pois a pessoa que o preencheu não fazia parte do grupo de desenvolvedores e consequentemente do grupo de sujeitos da pesquisa. Assim, na análise de dados foram considerados apenas onze questionários respondidos. Com base nos dados obtidos após a aplicação dos questionários, foram elaborados os gráficos apresentados a seguir, que possibilitam uma análise

da opinião dos desenvolvedores acerca da utilização do impacto da adoção de AspectJ para customização do SIG na UFS.

Com base na primeira pergunta do questionário, foi elaborado o gráfico da figura 6.8. Esse gráfico, apresenta os dados referentes à formação dos desenvolvedores que trabalham atualmente na manutenção do SIG. É possível observar que, entre os onze participantes há dois desenvolvedores com curso de graduação completo e dois cursando a graduação, dois com especialização completa e dois com curso em andamento, além de dois desenvolvedores com mestrado completo e um cursando.

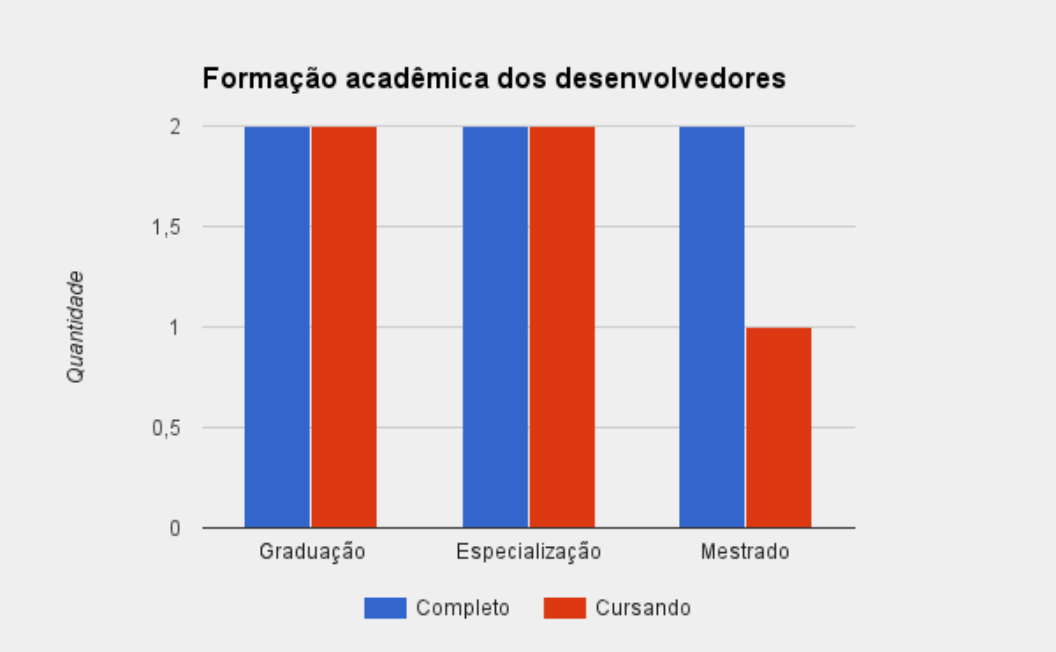


Figura 6.8: Formação acadêmica dos desenvolvedores

Para avaliar a experiência dos participantes da pesquisa com desenvolvimento em Java, foi questionado o tipo de experiência, se em sala de aula ou no trabalho e há quanto tempo possuem experiência com a linguagem. Observando o gráfico da figura 6.9, é possível identificar que a maioria já possui experiência no ambiente de trabalho e que 28,6% também afirmou ter experiência em sala de aula. A respeito do tempo de experiência com desenvolvimento em Java, observa-se no gráfico da figura 6.10, que a maioria trabalha com a linguagem há mais de 5 anos (5 dos participantes) ou entre 1 e 5 anos (4 dos participantes), e apenas dois participantes possuem menos de 6 meses de experiência e menos de 1 ano de experiência, respectivamente.

Figura 6.9: Experiência dos desenvolvedores com programação em Java

Figura 6.10: Tempo de experiência dos desenvolvedores com programação em Java

Foi avaliado também o tempo de experiência dos desenvolvedores com o trabalho na manutenção do SIG na UFS. Os dados são apresentados na figura 6.11. Nesse gráfico, é possível observar que mais da metade dos participantes (54,5%) possui entre 1 e 5 anos de experiência com a abordagem atual de manutenção adotada pela UFS.

Figura 6.11: Tempo de trabalho na manutenção do SIG

Após avaliar a formação e a experiência dos desenvolvedores, foi questionada a opinião dos participantes acerca da adequação da abordagem atual adotada pela UFS para customização e atualização do SIG. No gráfico da figura 6.12, apresenta-se a opinião dos participantes sobre a adequação da abordagem atual para introdução de variações no SIG. Observa-se que apenas 27,3% dos participantes consideram a abordagem adequada e que a maior parte a considera parcialmente adequada (54,5%). Em relação a atualização de versões com a abordagem atual, 72,7% dos participantes consideram-na parcialmente adequada. Com esses dados, é possível identificar que a maior parte dos desenvolvedores com experiência na abordagem atual de desenvolvimento a consideram parcialmente adequada para lidar com as variações introduzidas pela UFS no SIG e para a atualização de versões.

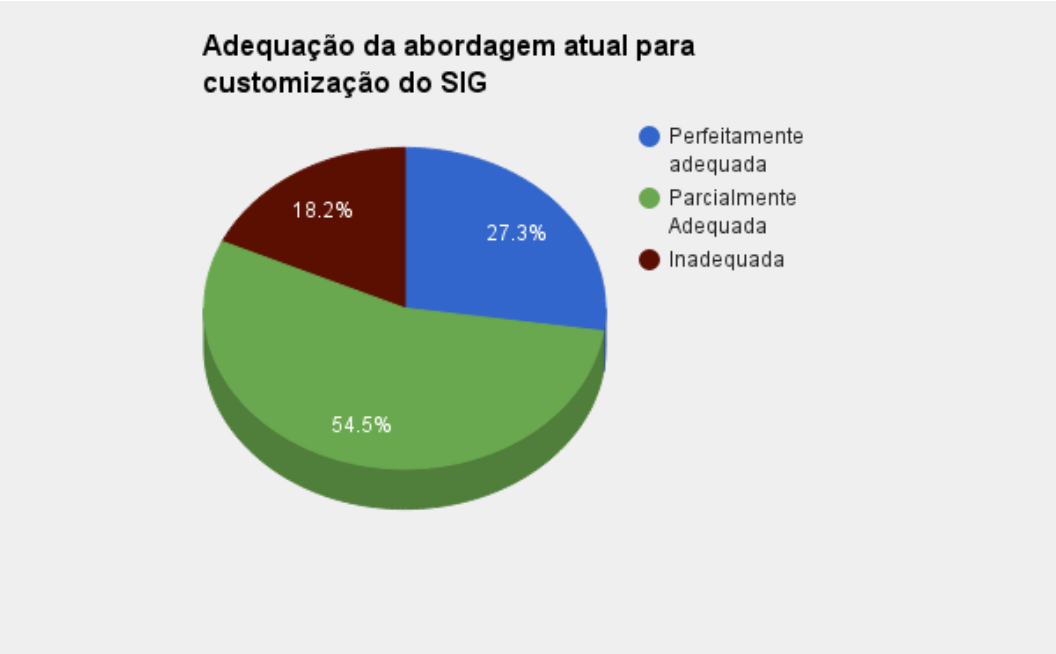


Figura 6.12: Adequação da abordagem atual adotada pela UFS para customização do SIG

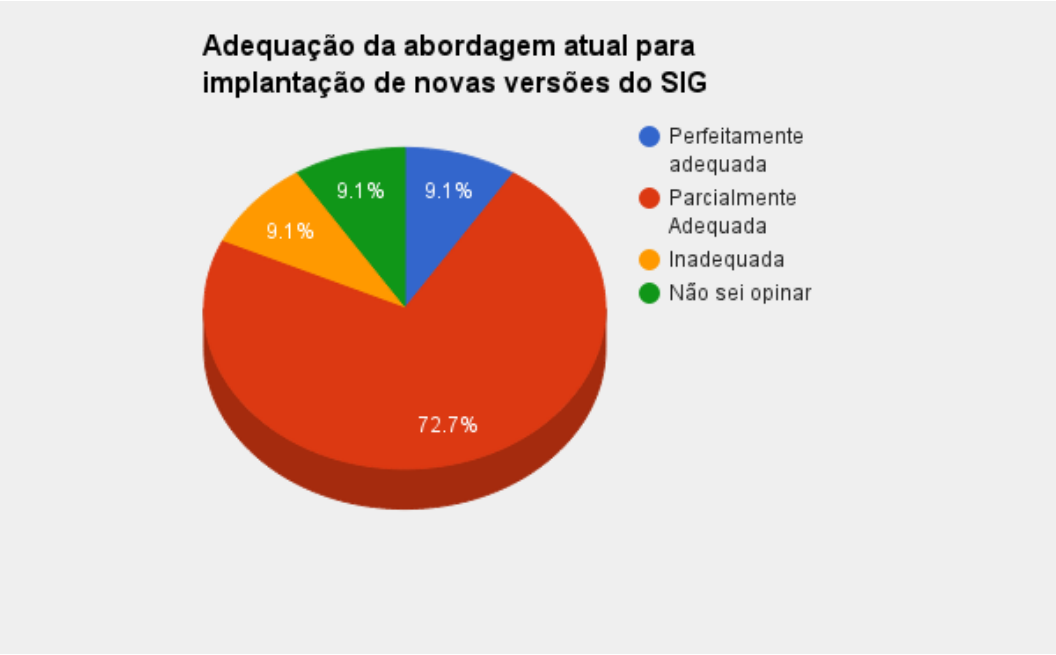


Figura 6.13: Adequação da abordagem atual para implantação de novas versões do SIG, ou seja, trazer uma nova versão do SIG e incluir as modificações requeridas pela UFS

Em seguida, foi avaliada a a experiência dos desenvolvedores com AspectJ e há quanto tempo possuem experiência com a linguagem. Dentre os 11 participantes, como é possível visualizar no gráfico da figura 6.15, apenas 2 desenvolvedores informaram ter menos de 1

ano e menos de 6 meses de experiência com a linguagem e apenas em sala de aula. Como é possível observar no gráfico da figura 6.14, 81,8% dos participantes não possuem experiência com AspectJ. Esse resultado, demonstra a necessidade de aplicar um treinamento adequado para a equipe conhecer a linguagem, caso a abordagem proposta venha a ser utilizada.

Figura 6.14: Experiência com desenvolvimento em AspectJ

Figura 6.15: Tempo de experiência com desenvolvimento em AspectJ

Em seguida, foi avaliada a opinião dos participantes da pesquisa em relação à adoção da abordagem proposta utilizando AspectJ, dentro do ambiente de desenvolvimento da UFS. Foi questionada a opinião dos desenvolvedores acerca do nível de complexidade implementar as variações (customizações) requeridas pela UFS no SIG usando AspectJ e, como é possível observar visualizando-se o gráfico da figura 6.16, 45,5% dos participantes consideraram um nível médio de complexidade e 27,3% considerou um nível alto, enquanto apenas 9,1% considerou um nível baixo de complexidade e 18,2% se absteve de opinar.

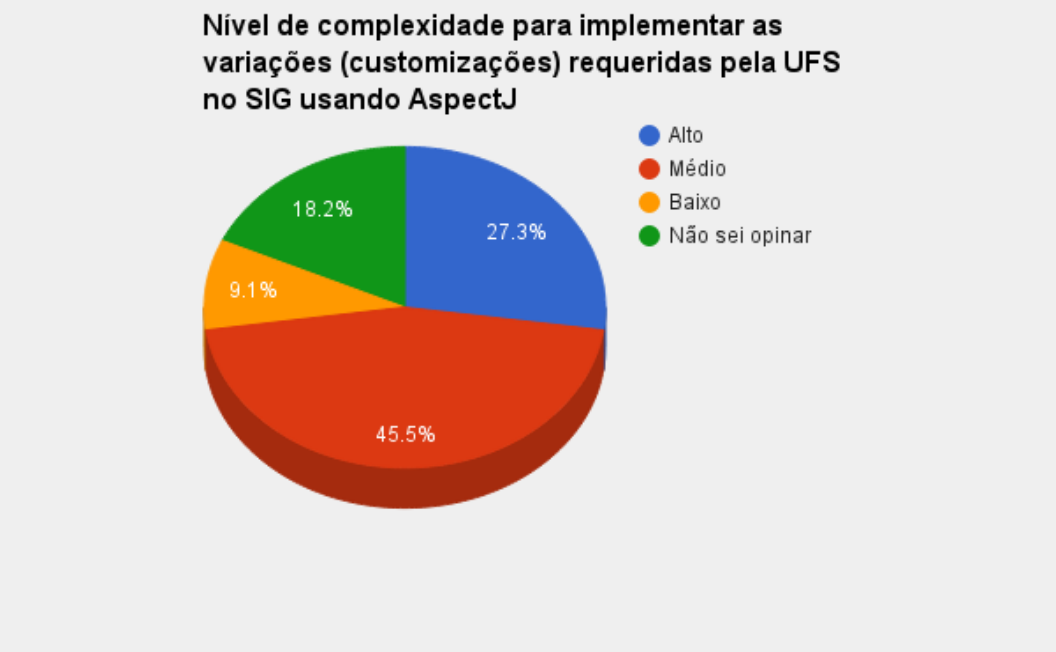


Figura 6.16: Nível de complexidade para implementar as variações (customizações) requeridas pela UFS no SIG usando AspectJ

Foi questionada a opinião dos participantes sobre a adequação de AspectJ para implantação de novas versões do SIG e 63,6% considerou-a parcialmente adequada, enquanto 9,1% a considerou perfeitamente adequada e 27,3% preferiu não opinar. Os dados referentes a essa pergunta estão disponíveis no gráfico da figura 6.17. Em relação à possibilidade de AspectJ diminuir o tempo necessário para a atualização de versões do SIG em relação à abordagem atual, 45,5% informaram que talvez seja possível, 27,3% acreditam que seja possível, enquanto 18,2% não souberam opinar e 9,1% consideraram que não acredita na possibilidade. No gráfico da figura 6.18 estão disponíveis os dados referentes a esta pergunta.

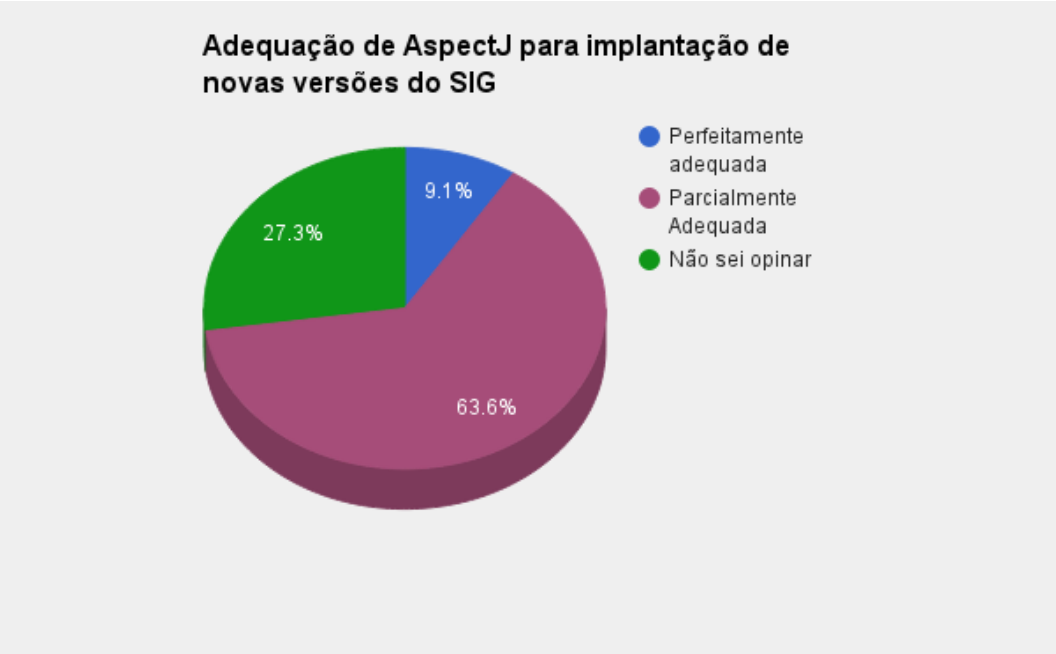


Figura 6.17: Opinião sobre a adequação de AspectJ para implantação de novas versões do SIG

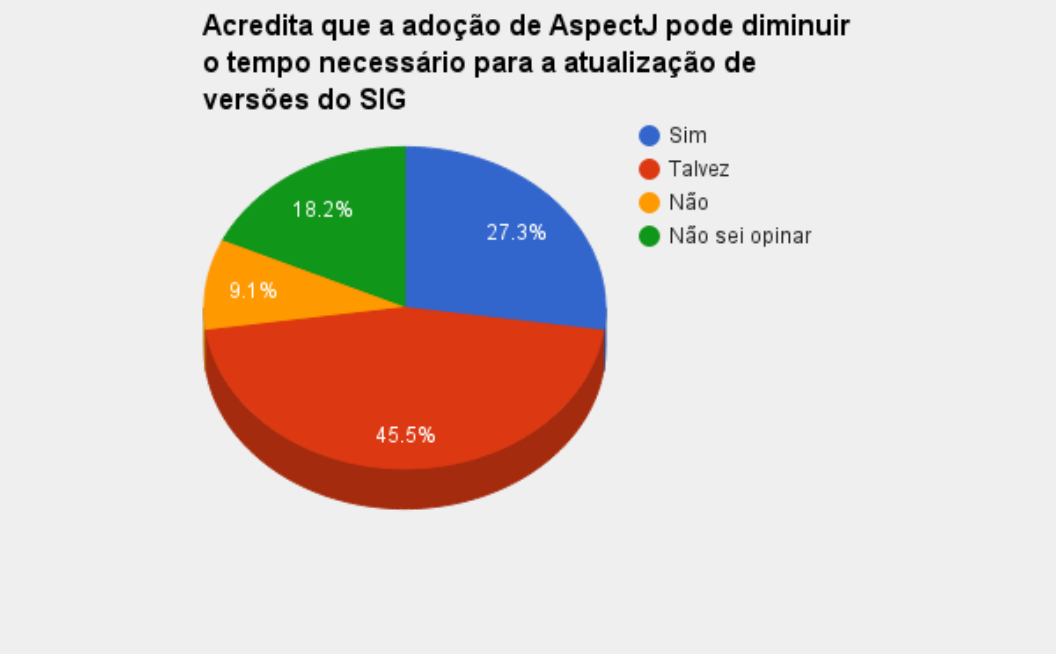


Figura 6.18: Opinião sobre a adoção de AspectJ diminuir o tempo necessário para a atualização de versões do SIG

Por fim, foi questionado aos participantes quais obstáculos podem ser considerados de grande, médio e pequeno impacto para adoção de AspectJ no ambiente da UFS, os dados

dessa pergunta estão disponíveis no gráfico da figura 6.19. O tempo adicional de treinamento foi considerado por 3 participantes de grande impacto, por 5 participantes foi considerado de médio impacto e por 2 de pequeno impacto, enquanto 1 participante preferiu não opinar. Sobre o domínio da linguagem, 6 participantes consideraram um nível médio de impacto, enquanto 4 consideram de pequeno impacto e 1 preferiu não opinar. Sobre a complexidade de implementação das variações, 7 participantes consideraram de médio impacto e 3 de pequeno impacto, enquanto 1 preferiu não opinar. Em relação ao aumento do tempo necessário para fazer o *build*, foi informado aos desenvolvedores sobre a observação durante a implementação das variações a respeito do aumento no tempo de compilação, mas que os resultados eram apenas empíricos, não havendo nenhum estudo aprofundado a respeito das causas e como solucionar esse problema, mas como esse obstáculo pode afetar o desempenho dos desenvolvedores foi apresentado o questionamento a respeito desse obstáculo. Dentre os participantes, 6 consideraram um obstaculo de médio impacto, 2 consideraram de grande impacto e 2 de pequeno impacto, enquanto 1 dos participantes preferiu não opinar.

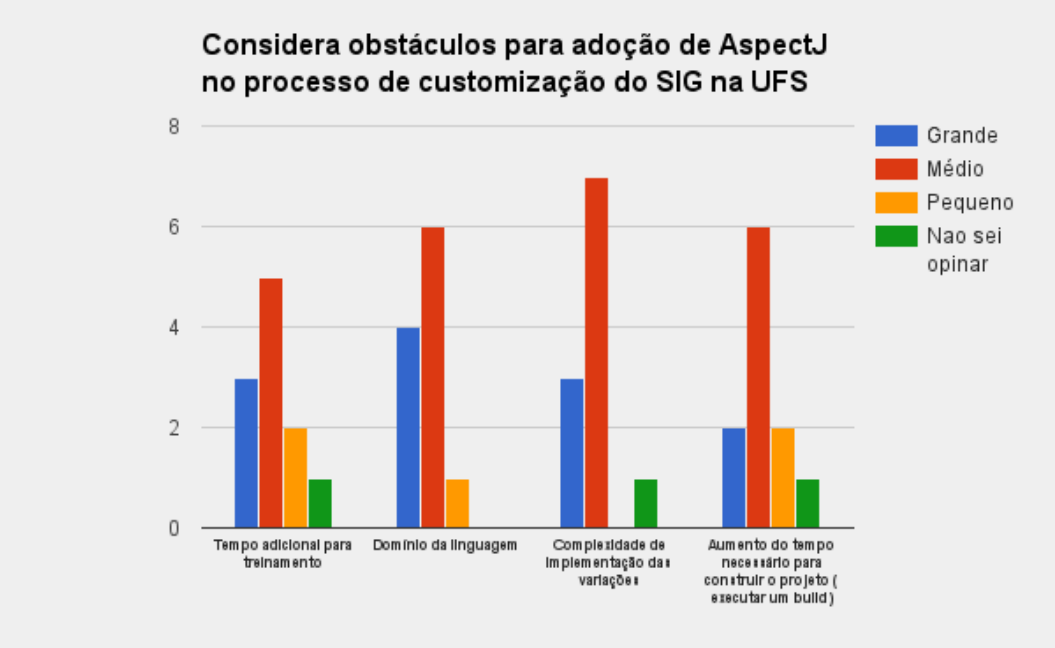


Figura 6.19: Opinião sobre os obstáculos para a adoção de AspectJ na manutenção do SIG

Capítulo 7

Conclusão

Neste trabalho de mestrado foi apresentada uma avaliação do impacto da adoção da Programação Orientada a Aspectos sobre o processo de manutenção e evolução de sistemas integrados de gestão, por meio de um estudo de caso conduzido no Núcleo de Tecnologia da Informação na Universidade Federal de Sergipe.

O trabalho foi conduzido em três etapas que envolveram o levantamento de variações introduzidas pela UFS utilizando a abordagem de desenvolvimento atual usada pelos desenvolvedores do NTI no módulo de Produção Intelectual do SIGAA; a implementação e avaliação das variações utilizando a POA e por fim a atualização de versão do módulo reintroduzindo-se as variações implementadas na etapa anterior.

Como resultado do trabalho, foi observado que, utilizando a POA, foi possível implementar 99,68% das variações encontradas. Para implementar algumas variações (6,75%), foram necessárias algumas adaptações que não alteraram o comportamento esperado das classes alvo das modificações. A atualização de versão do módulo demonstrou a ocorrência de seis erros nos Aspectos implementados, dois desses erros precisam ser avaliados cuidadosamente e requerem uma decisão de projeto na modificação do código base ou das variações introduzidas pela UFS, diante da evolução das classes alvo da variação implementada e da ocorrência de conflitos.

A quantidade de variações que foram implementadas com a adoção da POA e o pequeno número de erros detectados na etapa de atualização do módulo realizada nesse estudo de

caso, são alguns dos benefícios na utilização da Programação Orientada a Aspectos do ponto de vista da introdução de variações no SIG, no contexto de da UFS e possivelmente de outras instituições adquirentes do referido sistema. Porém, a maior complexidade para implementação das variações com AspectJ, o aumento no consumo de memória e no tempo de compilação do projeto observados durante o desenvolvimento da implementação, demonstram que possivelmente uma técnica de adaptação híbrida seja mais viável do ponto de vista de codificação. Há ainda que se considerar o impacto necessário para o treinamento da equipe de desenvolvimento e até o domínio da linguagem para a implementação de variações mais complexas. Considerando que atualmente a equipe de desenvolvimento não conhece o suficiente da linguagem para adotá-la no desenvolvimento, conforme observado na aplicação do questionário.

Porém, como foi descrito na seção 6.1, o processo atual de desenvolvimento adotado pela UFS, introduz uma série de problemas no código fonte que comprometem as possibilidades de continuidade da manutenção do sistema e de evolução do mesmo, ainda de acordo com os desenvolvedores que responderam o questionário aplicado

Como possibilidades de trabalhos futuros, verifica-se a necessidade de investigar o problema observado em relação à proporção de Aspectos implementados e a elevação no consumo de memória e tempo de compilação do projeto, já que foi apontado pelos desenvolvedores do SIG como um obstáculo ao uso da POA. Deve-se considerar ainda que neste estudo de caso foram implementadas as variações de apenas um módulo e que seria implementado um número ainda maior de Aspectos, caso as variações introduzidas em todos os módulos do SIGAA fossem aplicadas com essa técnica.

Outra possibilidade de estudo é a implementação de novos requisitos de outros módulos enviados para UFS com AspectJ e a medição de atributos como tempo e eficiência para implementação. Devido ao número pequeno de desenvolvedores e analistas bem como ao alto número de demandas de manutenção, a equipe de desenvolvimento pode ter dificuldades para investir tempo e pessoal na utilização de AspectJ dentro do ambiente de produção a abordagem atual. Por isso, existe também a possibilidade de conduzir um estudo de caso da implantação do módulo de produção intelectual com as variações implementadas em AspectJ no ambiente de produção.

Existe ainda a possibilidade de investigar os problemas detectados em relação ao tempo de compilação e consumo de memória elevados durante o *weaving* do projeto. Estudos demonstram que a adoção da POA pode causar um impacto no desempenho de aplicações durante a execução apenas quando o número de combinações de *pointcuts* é muito grande (SILVA; MAIA; SOARES, 2014; SOARES; MAIA; SILVA, 2015). Esta também é uma possibilidade a ser investigada.

Além disso, destaca-se a necessidade de apoio às variações implementadas, já que após a atualização foi necessário conferir todas as implementações realizadas e verificar a existência de erros. Esse processo foi realizado por meio de uma inspeção manual, em cada Aspecto. Contudo, a fim de oferecer suporte à essa atividade, é possível utilizar uma Linguagem para Especificação de Regras de Projeto (NETO *et al.*, 2013). Nesse contexto, a adoção LSD poderia melhorar a confiança do processo de introdução das variações em uma atualização, pela especificação das variações implementadas que poderia ser verificada a cada atualização. As restrições especificadas com a LSD possibilitariam dessa forma uma evolução independente das classes e aspectos, facilitando ainda mais o processo de atualização do SIG. Como possibilidade de estudo futuro, poderia ser avaliada, nesse contexto, a adoção da LSD.

Os resultados obtidos nesse estudo de caso não podem ser generalizados para outros cenários de introdução de variações, uma vez que o trabalho realizado avalia apenas o cenário do SIG.

Apêndice A - Questionário de avaliação

Avaliação da Adoção Programação Orientada a Aspectos para Melhoria do Processo de Manutenção e Evolução de Sistemas Integrados de Gestão

1. Qual a sua formação acadêmica?
    - Ensino médio: ( ) completo ( ) cursando
    - Graduação: ( ) completo ( ) cursando
    - Especialização: ( ) completo ( ) cursando
    - Mestrado: ( ) completo ( ) cursando
2. Você possui experiência com desenvolvimento em Java?
    - ( ) Em sala de aula
    - ( ) No trabalho
    - ( ) Nenhuma
3. Em caso positivo, há quanto tempo você programa em Java?
    - ( ) Menos de 6 meses
    - ( ) Entre 6 meses e 1 ano
    - ( ) Mais de 1 ano
    - ( ) Entre 1 e 5 anos
    - ( ) Mais de 5 anos

4. Há quanto tempo você trabalha na manutenção/desenvolvimento do SIG?

    - ( ) Menos de 6 meses
    - ( ) Entre 6 meses e 1 ano
    - ( ) Mais de 1 ano
    - ( ) Entre 1 e 5 anos
    - ( ) Mais de 5 anos

5. Quão adequada você considera a abordagem atual de manutenção da versão customizada pela UFS do SIG?

    - ( ) Perfeitamente adequada
    - ( ) Parcialmente Adequada
    - ( ) Inadequada
    - ( ) Não sei opinar

6. Quão adequada você considera a abordagem atual para implantação de novas versões do SIG, ou seja, trazer uma nova versão do SIG e incluir as modificações requeridas pela UFS?

    - ( ) Perfeitamente adequada
    - ( ) Parcialmente Adequada
    - ( ) Inadequada
    - ( ) Não sei opinar

7. Você possui experiência com desenvolvimento em AspectJ?

    - ( ) Em sala de aula
    - ( ) No trabalho
    - ( ) Nenhuma

8. Em caso positivo, há quanto tempo você programa em AspectJ?

    - ( ) Menos de 6 meses

- ( ) Entre 6 meses e 1 ano
- ( ) Mais de 1 ano
- ( ) Entre 1 e 5 anos
- ( ) Mais de 5 anos

9. Na sua opinião, qual é o nível de complexidade para implementar as variações (customizações) requeridas pela UFS no SIG usando AspectJ?

   - ( ) Alto
   - ( ) Médio
   - ( ) Baixo
   - ( ) Não sei opinar

10. Quão adequada você considera AspectJ para implantação de novas versões do SIG, isto é, trazer uma nova versão do SIG e incluir todas as modificações requeridas pela UFS?

    - ( ) Perfeitamente adequado
    - ( ) Parcialmente Adequado
    - ( ) Inadequado
    - ( ) Não sei opinar

11. Você acredita que a adoção de AspectJ pode diminuir o tempo necessário para a atualização de versões do SIG, isto é, mantê-lo atualizado com a versão mais recente e conter as modificações requeridas pela UFS?

    - ( ) Sim
    - ( ) Talvez
    - ( ) Não
    - ( ) Não sei opinar

12. Quais desses pontos você considera obstáculos para a adoção de AspectJ na manutenção do SIG?

    - Tempo adicional para treinamento:

      ( ) Grande ( ) Médio ( ) Pequeno ( ) Não sei opinar

    - Domínio da linguagem:

      ( ) Grande ( ) Médio ( ) Pequeno ( ) Não sei opinar

    - Complexidade das implementações das variações:

      ( ) Grande ( ) Médio ( ) Pequeno ( ) Não sei opinar

    - Aumento do tempo necessário para construir o projeto (executar um build):

      ( ) Grande ( ) Médio ( ) Pequeno ( ) Não sei opinar

Referências Bibliográficas

ALUR, D.; CRUPI, J.; MALKS, D. *Core J2EE Patterns: Best Practices and Design Strategies*. [S.l.]: Prentice Hall Ptr, 2003.

ALVES, V. *et al.* From conditional compilation to aspects: a case study in software product lines migration. 2006.

ALVES, V. R. *Implementing software product line adoption strategies*. Tese (Doutorado) — UFPE, 2007.

APEL, S.; KÄSTNER, C. Virtual separation of concerns - A second chance for preprocessors. *Journal of Object Technology*, v. 8, n. 6, p. 59–78, 2009. Disponível em: <http://dx.doi.org/10.5381/jot.2009.8.6.c5>.

BASILI, V. R. *et al.* Goal question metric (gqm) approach. *J. Marciniak: Encyclopedia of Software Engineering*, v. 1, p. 578–583, 2002.

BRAGA, R. T. V. *et al.* Aiple-is: An approach to develop product lines for information systems using aspects. In: *Proceedings of Brazilian Symposium on Software Components, Architectures, and Reuse*. [S.l.: s.n.], 2007.

CLEMENTS, P.; NORTHROP, L. M. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70332-7.

DURSCKI, R. C. *et al.* Linhas de produto de software: riscos e vantagens de sua implantação. *Simpósio Brasileiro de Processo de Software*, 2004.

EASTERBROOK, S. *et al.* Selecting empirical methods for software engineering research. In: *Guide to advanced empirical software engineering*. [S.l.]: Springer, 2008. p. 285–311.

ECLIPSE. *Introduction to AspectJ - Programming Guide*. 06 2015. Disponível em: <https://eclipse.org/aspectj/doc/released/progguide/starting-aspectj.htmlthe-dynamic-join-point-model>.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 29–32, out. 2001. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/383845.383853>.

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

GAMMA, E. *et al. Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1995.

GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. São Paulo: [s.n.], 2002. 175 p p.

GURP, J. van; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. In: *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01*. [S.l.]: IEEE Computer Society, 2001. p. 45–54.

JACOBSON, I.; GRISS, M.; JONSSON, P. *Software reuse: architecture process and organisation for business success*. New York: ACM Press, 1997.

KICZALES, G. *et al.* An overview of aspectj. In: . [S.l.]: Springer-Verlag, 2001. p. 327–353.

KICZALES, G. *et al.* Aspect-oriented programming. In: *ECOOP*. [S.l.]: SpringerVerlag, 1997.

LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications Co., 2003. ISBN 1930110936.

LAUDON, K. C.; TRAVER, C. G. *Management Information Systems*. 12th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN 9780132142854.

LIMA, G. *et al.* A delta oriented approach to the evolution and reconciliation of enterprise software products lines. In: HAMMOUDI, S. *et al.* (Ed.). *ICEIS (1)*. [S.l.]: SciTePress, 2013. p. 255–263. ISBN 978-989-8565-59-4.

MENS, T. *et al.* Challenges in software evolution. In: IEEE. *Principles of Software Evolution, Eighth International Workshop on*. [S.l.], 2005. p. 13–22.

MEYER, B. *Object-Oriented Software Construction (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN 0-13-629155-4.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011.

NETO, A. C. *et al.* A design rule language for aspect-oriented programming. *Journal of Systems and Software*, v. 86, n. 9, p. 2333 – 2356, 2013. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121213000861>.

OLIVEIRA, S. L. d. *Tratado de metodologia científica: projetos de pesquisas, TGI, TCC, monografias, dissertações e teses.* [S.l.: s.n.], 1998. 320 p p.

ORAM, A.; WILSON, G. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. [S.l.]: O'Reilly Media, Inc., 2007. ISBN 0596510047.

PAGLIARI, L. F.; NUNES, D. J. Um processo para avaliaç ao quantitativa de refatoraç oes de software. 2007.

PASSOS, F. *et al.* Experimentando aspectj como uma abordagem para lidar com a evoluçao e customizaçao de um sistema integrado de gestao. In: *WMod2014-11th Workshop on Software Modularity, Maceió, Brasil*. [S.l.: s.n.], 2014.

PASSOS, F. A.; NETO, A. C. 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*. [S.l.: s.n.], 2012.

PASSOS, F. A.; NETO, J. S. A. C. Adaptação e manutenção de sistemas integrados de gestão apoiados pela programação orientada a aspectos. In: *In Proceedings of The IX Simpósio Brasileiro de Sistemas de Informação - SBSI 2013*. [S.l.: s.n.], 2013.

POHL, K.; BÖCKLE, G.; LINDEN, F. V. D. *Software product line engineering: foundations, principles, and techniques*. [S.l.]: Springer, 2005.

PRESSMAN, R. S. *Engenharia de software*. [S.l.]: McGraw Hill Brasil, 2011.

RUBIN, J.; CZARNECKI, K.; CHECHIK, M. Managing cloned variants: A framework and experience. In: *Proceedings of the 17th International Software Product Line Conference*. New York, NY, USA: ACM, 2013. (SPLC '13), p. 101–110. ISBN 978-1-4503-1968-3. Disponível em: <http://doi.acm.org/10.1145/2491627.2491644>.

RUBIN, J. *et al.* Managing forked product variants. In: ACM. *16th International Software Product Line Conference (SPLC 2012)*. Salvador, Brazil: ACM, 2012.

RUNESON, P. *et al. Case study research in software engineering: Guidelines and examples*. [S.l.]: John Wiley & Sons, 2012.

SANTOS, J.; KULESZA, U. Avaliação de conflitos de merge de código em linhas de produtos de software clonadas. In: *WMOD2014*. [S.l.: s.n.], 2014.

SANTOS, J. *et al.* Execução condicional: Um padrão para implementação de variabilidades de granularidade fina. In: *Proceedings of 9th LatinAmerican Conference on Pattern Languages of Programming (SugarLoafPLoP 2012)*. [S.l.: s.n.], 2012.

SANTOS, L. E. d. S. *Processo de Teste de Software*. 1. ed. www.ufs.br, 1 2015.

SENA, D. *et al.* Modularizando variabilidades em linhas de produto de sistemas de informação web. In: *In proceedings os The 9th Latin American Conference on Pattern Languages of Programming*. [S.l.: s.n.], 2012.

SILVA, R. F. G. da; MAIA, M. de A.; SOARES, M. S. A systematic review on performance evaluation of aspect-oriented programming techniques used to implement crosscutting concerns. In: *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems, Volume 2, Lisbon, Portugal, 27-30 April, 2014*. [S.l.: s.n.], 2014. p. 5–13.

SINFO/UFRN. *Sistemas Institucionais Integrados de Gestão - SIG*. Novembro 2013. Disponível em: <http://www.info.ufrn.br/wikisistemas/doku.php>.

SOARES, M.; MAIA, M.; SILVA, R. Performance evaluation of aspect-oriented programming weavers. In: CORDEIRO, J. *et al.* (Ed.). *Enterprise Information Systems*. [S.l.]: Springer International Publishing, 2015, (Lecture Notes in Business Information Processing, v. 227). p. 187–203. ISBN 978-3-319-22347-6.

TRAVASSOS, G. H.; GUROV, D. *Introdução à engenharia de software experimental*. [S.l.: s.n.], 2002.

VENTURA, M. M. O estudo de caso como modalidade de pesquisa. *Revista SoCERJ*, v. 20, n. 5, p. 383–386, 2007.

YIN, R. K. *Estudo de Caso-: Planejamento e Métodos*. [S.l.]: Bookman editora, 2015.