UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos de Software Clonadas

Gleydson de Azevedo Ferreira Lima

Orientador: Prof. Dr. Sergio Viana Fialho

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica e de Computação.

Número de Ordem PPgEEC: D115

Natal, RN, Março de 2014

UFRN / Biblioteca Central Zila Mamede.

Catalogação da Publicação na Fonte.

Lima, Gleydson de Azevedo Ferreira.

Uma abordagem para evolução e reconciliação de linhas de produtos de software clonadas. / Gleydson de Azevedo Ferreira Lima. – Natal, RN, 2014.
186 f.: il.

Orientador: Prof. Dr. Sergio Viana Fialho.

Tese (Doutorado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e Computação.

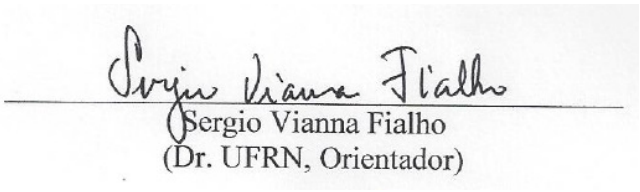
1. Engenharia de Linhas de Produto de Software - Tese. 2. Clonagem de Linhas de Produto de Software - Tese. 3. Conflitos de código - Tese. 4. Evolução de Software - Tese. 5. Mineração de repositório de Software - Tese. 6. Linhas de Produto para Sistemas de Informações Web - Tese. I. Fialho, Sergio Viana. III. Universidade Federal do Rio Grande do Norte. IV. Título.

RN/UF/BCZM CDU 004.4

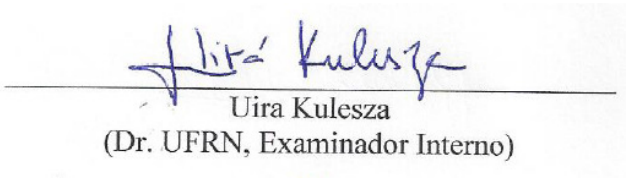
Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos de Software Clonadas

Gleydson de Azevedo Ferreira Lima

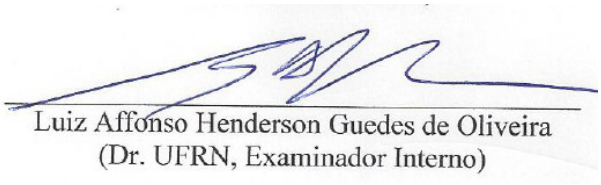
Defesa de Doutorado aprovada em 31 de Março de 2014 pela banca examinadora composta pelos seguintes membros:



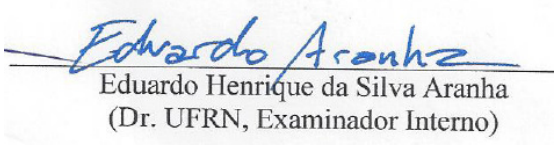
Sergio Vianna Fialho
(Dr. UFRN, Orientador)



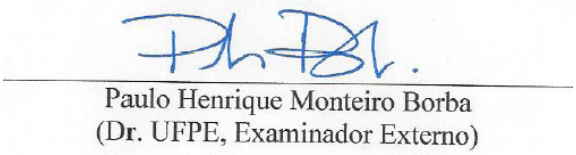
Uira Kulesza
(Dr. UFRN, Examinador Interno)



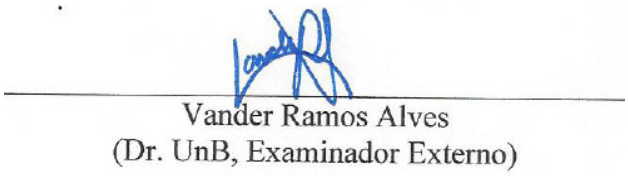
Luiz Affonso Henderson Guedes de Oliveira
(Dr. UFRN, Examinador Interno)



Eduardo Henrique da Silva Aranha
(Dr. UFRN, Examinador Interno)



Paulo Henrique Monteiro Borba
(Dr. UFPE, Examinador Externo)



Vander Ramos Alves
(Dr. UnB, Examinador Externo)

Agradecimentos

Agradeço a minha família pelo apoio sempre dedicado, em especial a minha esposa Raphaela Lima, cujo companheirismo, carinho e incentivo foram essenciais para o desenvolvimento deste trabalho. Ao meu pai, Djahy, por ter incentivado o meu gosto pelo estudo, especialmente pela tecnologia da informação desde que eram bem jovem. A minha mãe, Edna, pelo amor incondicional em todos os momentos.

Ao meu orientador Sérgio Fialho e ao Prof. Uirá Kulesza pelos ensinamentos, orientações e conselhos que foram fundamentais para a finalização desta tese. Ao Prof. Ivonildo Rego pelo apoio e motivação: sua visão estratégica-progressista incentivou a concretização de um doutorado na empresa. Aos professores Rubens Maribondo e Apuena Gomes pela torcida sincera pelo meu sucesso. A todos os demais amigos que que também torceram e me felicitaram por esta conquista.

Aos colegas Jadson Santos e Daniel Alencar pela ajuda na implementação da ferramenta desta tese. A todos os amigos, primos, familiares e colegas de trabalho da família SIG Software que apoiaram direta ou indiretamente este trabalho.

Resumo

Linhas de produtos de software promovem a reutilização em larga escala através do desenvolvimento de famílias de sistemas que: (i) compartilham um núcleo comum de características previamente implementadas; e (ii) permitem a seleção e customização das características variáveis, as quais determinam os comportamentos distintos de cada membro ou produto da família de sistema. Por razões de *time-to-market* e flexibilidade, a indústria de software tem adotado, com frequência, a técnica de clonagem como mecanismo de criação de produtos ou de novas linhas de produtos. Apesar das suas vantagens, a técnica de clonagem traz dificuldades para a evolução e reconciliação de características de linhas de produto de software devido aos possíveis conflitos de integração das mudanças realizadas no código da linha de produto de software original, denominada *Source*, e a da linha de produto clonada, denominada *Target*. Esta tese de doutorado propõe uma abordagem para evolução e reconciliação de produtos clonados pela adoção de técnicas de mineração de repositórios de software. A abordagem promove a identificação de diferentes tipos de conflitos – léxicos, estruturais e semânticos - que podem ocorrer durante a integração de características ou tarefas de desenvolvimento a partir da linha de produto de software original para linhas de produto clonadas. O trabalho apresenta os resultados de um estudo empírico de caracterização dos tipos de conflitos de integração de código em diferentes evoluções de duas linhas de produto de software de sistemas de informação Web de larga escala. Os resultados do estudo demonstram o potencial da abordagem na resolução automatizada ou semi-automatizada de vários dos conflitos existentes, reduzindo assim os custos e complexidade de evolução e reconciliação de linhas de produto de software clonadas**.**

Palavras Chaves: Engenharia de Linhas de Produto de Software, Clonagem de Linhas de Produto de Software, Conflitos de código, Evolução de Software, Mineração de repositório de Software, Linhas de Produto para Sistemas de Informações Web.

Abstract

Software product line engineering promotes large software reuse by developing a system family that shares a set of developed core features, and enables the selection and customization of a set of variabilities that distinguish each software product family from the others. In order to address the time-to-market, the software industry has been using the clone-and-own technique to create and manage new software products or product lines. Despite its advantages, the clone-and-own approach brings several difficulties for the evolution and reconciliation of the software product lines, especially because of the code conflicts generated by the simultaneous evolution of the original software product line, called *Source*, and its cloned products, called *Target*. This thesis proposes an approach to evolve and reconcile cloned products based on mining software repositories and code conflict analysis techniques. The approach provides support to the identification of different kinds of code conflicts – lexical, structural and semantics – that can occur during development task integration – bug correction, enhancements and new use cases – from the original evolved software product line to the cloned product line. We have also conducted an empirical study of characterization of the code conflicts produced during the evolution and merging of two large-scale web information system product lines. The results of our study demonstrate the approach potential to automatically or semi-automatically solve several existing code conflicts thus contributing to reduce the complexity and costs of the reconciliation of cloned software product lines.

Keywords: Software product line engineering, clone-and-own approach, software merge, code conflicts, software evolution, mining software repository, web information system software product line.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Desde a década de 70, quando a crise do software foi deflagrada, a Ciência da Computação busca alternativas para o desenvolvimento de software através de técnicas de engenharia que tornem o processo sustentável e de alta qualidade. A engenharia de software tornou-se então uma área de grande investigação e investimentos, para que a complexidade do desenvolvimento de sistemas possa ser enfrentada com metodologias estruturadas e sistemáticas.

As técnicas da engenharia de software tradicionais se baseiam no desenvolvimento de um produto de software para um determinado contexto de negócio. Ou seja, aplica-se a engenharia de requisitos para o estudo de um determinado domínio e regras do negócio dentro de um escopo de problema ou organização. Em seguida, segue-se um processo de engenharia para as atividades de análise, projeto, implementação e testes daquele sistema, de forma a obter o software que solucione o escopo antes identificado. Neste contexto, o software desenvolvido possui características inerentes ao problema estudado, e dessa forma a sua adaptação para novos cenários é, em geral, restrita e nem sempre trivial, exigindo em muitos casos modificações invasivas e de granularidade fina no código do sistema originalmente criado, o que muitas vezes conduz a outros produtos distintos.

Outras áreas da indústria, tais como a automobilística, buscaram aprimorar suas técnicas de produção através do desenvolvimento de linhas de produto. Automóveis são produzidos com itens de série, presentes em todas as versões, e itens opcionais, que visam atender às diferentes necessidades e exigências dos clientes. Assim, desde sua concepção, o projeto prevê similaridades e variabilidades entre as várias versões de automóveis.

Esta necessidade não é exclusiva da indústria automobilística, situações similares ocorrem com o software. Um sistema com um determinado propósito de mercado, quando aplicado a realidades diferentes (ou clientes distintos) costuma exigir novas funcionalidades ou adaptações nas versões existentes. A falta de técnicas para lidar com tais mudanças pode inviabilizar a evolução deste sistema, criando assim

diferentes programas incompatíveis que podem conduzir a custos de manutenção exponenciais.

Pesquisas iniciadas na década de 90 introduziram o conceito de *Feature-Oriented Domain Analysis* (FODA) (Kang et al, 1990), cujo objetivo era criar um modelo de domínio (modelo de características, do inglês *Feature Model*) que represente uma família de sistemas que possa ser refinado para um produto em particular. Diversas empresas e pesquisas acadêmicas (Clements, 2002) avançaram no sentido de definir métodos para uma evolução robusta de software que possam ser derivados para uma vasta variedade de clientes que, ao mesmo tempo, compartilham características comuns, mas também possuem suas peculiaridades. Definiu-se então uma nova área da Engenharia de Software: a Linha de Produto de Software.

Uma linha de produto de software (LPS) (Clements & Northrop, 2001) propõe-se a minimizar os custos de desenvolvimento e evolução de produtos que fazem parte de uma família de sistemas. Uma família de programas (Parnas, 1976) consiste em um conjunto de sistemas que possuem várias funcionalidades em comum, mas que também mantém suas características individuais e específicas. Assim, dentre os programas que fazem parte da família existem um conjunto de características comuns (similaridades) e aquelas características que são inerentes de um produto ou de um subconjunto de produtos (variabilidades). Uma característica (do inglês, *feature*) representa uma funcionalidade ou propriedade da linha de produto que é relevante para algum de seus interessados.

Para o desenvolvimento de uma linha de produto de software é preciso que todas as etapas do processo de desenvolvimento promovam a gerência das similaridade e variabilidades. A identificação destas características no decorrer do processo deve ser feita em diversos artefatos (do inglês, *assets*), desde os requisitos à arquitetura como também da implementação ao teste. Esta coleção de artefatos, quando agrupadas, define a infraestrutura da LPS. Assim, o desenvolvimento de uma LPS envolve tipicamente duas etapas (Czarnecki & Eisenecker, 2000): (i) engenharia de domínio – onde são criados, projetados e implementados artefatos (requisitos, arquitetura, projeto, código e testes) que contemplam as similaridades e variabilidades da linha de produto; e (ii) engenharia de aplicação – que promove a criação de produtos específicos através do reuso e customização dos artefatos produzidos durante a engenharia de domínio.

A manutenção da infraestrutura da LPS exige mudanças no processo de desenvolvimento de software tradicional, para que as evoluções do software possam ser previamente analisadas, categorizadas como similaridade ou variabilidade, e assim direcionar corretamente à implementação, garantindo uma evolução robusta da LPS. Esta metodologia torna-se ainda mais desafiadora se consideramos o cenário onde times de desenvolvimento distintos, de diferentes organizações, evoluem a LPS de forma concomitante.

1.1 Contextualização do Problema

Ao longo dos últimos anos, diversas técnicas e mecanismos de projeto e implementação de arquiteturas de linhas de produto de software têm sido propostas e exploradas (Clements & Northrop, 2001) (Czarnecki & Eisenecker, 2000) (Greenfield & Short, 2005) (Weiss & Lai, 1999), com o objetivo de melhor modularizar suas variabilidades, facilitando assim a evolução para novos cenários e contextos. As técnicas propostas baseiam-se em cenários de incremento de funcionalidades bem definidas na linha, tendo o comportamento desejado na flexibilização da estrutura do software considerando variabilidades já previstas. Entretanto, no contexto de evoluções que ocorrem de forma descentralizada e em pontos não necessariamente previstos, tais técnicas podem não ser suficientes para evitar a separação dos produtos da linha e sua evolução com um ciclo de vida próprio.

Sistemas de informação Web de média e larga escala são excelentes candidatos a serem desenvolvidos como linhas de produto de software, pois normalmente são de grande porte, de alto escopo de funcionalidades e exigem constantes adaptações e parametrizações para atender aos diferentes contextos organizacionais. Os métodos e técnicas que representam o estado da arte de engenharia de LPS permitem que uma determinada empresa evolua seu sistema de informação Web para diferentes clientes, mantendo centralizada a gerência das características similares e variáveis e controlando todo o processo de evolução da LPS.

Neste contexto, é possível considerar o cenário onde uma organização desenvolve uma linha de produto de software, a distribui para diversas organizações

clientes, e permite que os times de engenheiros de software destas organizações evoluam a LPS de forma autônoma. Este cenário permite uma maior independência e flexibilidade de trabalho tanto para a empresa desenvolvedora da LPS original quanto para as demais empresas parceiras que customizam a LPS de acordo com suas necessidades, o que contribui para acelerar o processo de criação e evolução de cada uma das versões individuais. Por outro lado, acaba dificultado o processo de gerenciamento e manutenção das similaridades e variabilidades da LPS, já que devido a produção de diferentes versões, tal informação não se encontra mais centralizada em uma infraestrutura única de artefatos, e dessa forma, impossibilita a migração de funcionalidades desenvolvidas ou evoluídas de uma versão da LPS de uma empresa para outra.

A ausência de um controle centralizado de gerência de mudanças e variabilidades da LPS e a flexibilidade da evolução simultânea das diferentes versões da LPS geram consequências de negócio importantes, pois cada organização detém a prerrogativa de evoluir seu produto da maneira que lhe convier, impedindo assim que, por limitações técnicas, o cliente tenha que se moldar a comportamentos indesejados implementados na LPS original. Porém, incorpora a possibilidade da geração de diversos tipos de conflitos de código quando se está integrando as diferentes implementações das versões da LPS, o que torna a reconciliação das evoluções uma tarefa custosa e propensa a erros, se considerarmos apenas as técnicas atuais e disponíveis.

Em particular, nesta tese de doutorado, este problema é abordado dentro do contexto de desenvolvimento e evolução de linhas de produto de sistemas de informação Web, denominados Sistemas Integrados de Gestão (SIG), sob propriedade da Universidade Federal do Rio Grande do Norte (UFRN). Esta LPS é mantida e evoluída pela própria UFRN, ao mesmo tempo que diversas outras instituições clonam seus códigos em *branches* em um repositório de acesso compartilhado e então as customizam para a sua realidade. Também a empresa SIG Software e Consultoria em TI mantém e evolui estas LPS da UFRN considerando outras quinze instituições, sendo uma das colaboradoras desta pesquisa para análise das linhas de produtos clonadas.

Atualmente, diversas instituições clonam essas LPS de sistemas de informação Web estendendo variabilidades já pré-definidas no código, assim como realizam

customizações específicas. Tais customizações podem eventualmente ocasionar mudanças invasivas no núcleo da LPS. A migração de tarefas de desenvolvimento de novas funcionalidades, melhoria de funcionalidades existentes ou correção de *bugs* são realizadas manualmente apenas com o suporte de ferramentas de controle de versão e recursos de análise léxica de conflitos de classes modificadas simultaneamente em duas versões de uma mesma LPS. Como resultado, a clonagem das LPS tem levado à desistência da reconciliação entre versões clonadas de forma a agregar novas funcionalidades desenvolvidas pelos parceiros.

1.2 Limitações de Abordagens Existentes

A abordagem proposta por Krueger (Krueger, 2002) preocupa-se com os vários cenários de evolução da LPS caracterizando três estratégias de adoção de linhas de produto: proativa, reativa e extrativa. Na estratégia proativa as variabilidades são mapeadas desde o início do processo de desenvolvimento, seguindo o fluxo tradicional da engenharia de domínio e aplicação. Na abordagem reativa o núcleo da LPS cresce gradativamente à medida que novos produtos ou novos requisitos em produtos existentes surgem e são incorporados ao núcleo. Finalmente, a abordagem extrativa desenvolve a infraestrutura da LPS a partir das variabilidades e similaridades de produtos existentes. Essas estratégias de adoção de LPS necessitam de uma gestão mais centralizada para que as variabilidades obtidas em produtos possam ser incorporadas na infraestrutura da linha de produto de software. Tais abordagens também não se propõem a contribuir com a resolução dos problemas de conflitos de integração (*merge*) de código que por ventura possam se manifestar entre os diversos produtos clonados e evoluídos de uma mesma LPS.

O reuso de software através da técnica da clonagem tem sido, historicamente, uma das menos recomendadas devido à replicação de artefatos e as dificuldades geradas na gestão das diferentes versões, porém por ser um mecanismo simples e eficiente de reutilização, tem sido utilizado com frequência. O reuso por clonagem permite economia de tempo e recursos, ao mesmo tempo que proporciona independência e liberdade para mudar esses artefatos conforme a necessidade. Em um estudo recente, Dubinsky et al (Dubinsky, et al., 2013) pesquisaram o uso da clonagem em seis

empresas de diferentes áreas e concluíram que apesar dos desafios esta abordagem vem sendo adotada satisfatoriamente por tais empresas. Dentre os principais desafios destaca-se o gerenciamento da base de conhecimento da evolução dos clones, neste contexto, Rubin & Checkik (Rubin & Chechik, 2013) propõem um *framework* para gerenciamento de produtos clonados composto por sete operações conceituais para reconciliação de produtos. A abordagem proposta não se propõe a computar conflitos durante a reconciliação de funcionalidades entre produtos. Ela também carece de resultados concretos experimentais de reconciliação de *clones* de LPS, pois ainda são operações conceituais sem uma ferramenta de suporte e/ou validação da proposta. Rubin et al (Rubin et al, 2012) também propuseram uma abordagem para capturar as informações de evoluções dos clones em um modelo denominado PL-CDM: *Product Line Changeset Dependency Model.* Este modelo extraí informações de sistemas de gestão de demandas para agregar característica e calcular suas dependências. Tal modelo também não foi avaliado através de uma implementação concreta e condução de estudos empíricos.

Em paralelo às investigações relacionadas à clonagem, diversos grupos têm conduzido pesquisas focadas no problema da resolução de conflitos de integração de código quando equipes de desenvolvimento modificam e evoluem de forma paralela os mesmos artefatos. Brun et al (Brun et al., 2013) desenvolveram uma abordagem especulativa para tentar antecipar os conflitos antes deles se manifestarem, prevenindo que desenvolvedores publiquem seus códigos em conflitos com os de outros. Guimarães & Silva (Guimarães & Silva, 2012) propuseram uma abordagem de *merge* contínua que analisa os códigos publicados e não publicados em espaços de trabalhos locais e indica os diferentes tipos de conflitos antes da publicação do código. Esta abordagem também busca prevenir os conflitos antes deles serem publicados pelos desenvolvedores. Wloka et al (Wloka et al., 2009) propuseram uma abordagem para analisar o *software* em desenvolvimento e identificar mudanças que são passíveis de publicação no repositório de código através de diversas políticas de publicação sem causar erros de testes. Para determinar se os erros ocorrem ou não a abordagem extrai do modelo de mudanças as operações atômicas (adição, remoção e atualização de métodos e atributos) e os relaciona com o grafo de chamadas obtido através da execução dos testes unitários. Mens (Mens. T, 2002) analisou o estado da arte das operações de *merge*, categorizando-os quanto ao sentido do *merge*, a sua semântica e base de comparação. Zimmerman

(Zimmermann, 2007) avaliou a evolução de quatro sistemas de código aberto e concluiu que 23 a 46% dos cenários de *merge* resultam em conflitos no repositório CVS. Os trabalhos de pesquisa propostos e estudos empíricos conduzidos nesta área de resolução de conflitos de código não foram aplicados no contexto de LPS clonadas.

Outros trabalhos relacionados são os que abordam o desenvolvimento consciente (*awareness development*), ou seja, aqueles que permitem um desenvolvedor entender as atividades que estão sendo realizadas por outros desenvolvedores, proporcionando um contexto para suas próprias atividades (Dourish & Bellotti, 1992). Ferramentas foram também propostas em trabalhos de pesquisa recentes (Sarma et al., 2013) (Hattori &.; Lanza, 2010) com o objetivo de aumentar as informações do trabalho da equipe para membros individuais durante o desenvolvimento colaborativo e assim evitar que conflitos de códigos possam ocorrer.

Assim, a pesquisa de trabalhos relacionados a esta tese de doutorado identificou três áreas principais: (i) pesquisas relacionadas a conflitos de código no desenvolvimento em equipes e nas operações de *merge*; (ii) ferramentas que identificam conflitos e informações relevantes durante o desenvolvimento, e fornecem canais de comunicação para tratá-los; e (iii) estudos relacionados ao gerenciamento de clones em LPS. Os trabalhos relacionados a conflitos, em geral, são orientados a publicações em repositórios (*commits*), mas não abordam o problema dos clones em LPS. Já os trabalhos relacionados com clones em LPS não têm considerado a análise de conflitos da integração do código como parte das abordagens propostas. Além disso, foi identificado que existe uma ausência de estudos empíricos conduzidos na área de LPS clonadas.

1.3 Questões de Pesquisa

De acordo com a contextualização do problema desta tese as seguintes questões de pesquisa são propostas:

- É possível evoluir concomitantemente LPS de sistemas de informação Web, através da técnica da clonagem e, posteriormente, reconciliar as tarefas de evolução de cada LPS de forma independente?

- Que tipos de conflitos ocorrem quando se está realizando a integração e *merge* de tarefas de desenvolvimento de linhas de produto de software clonadas e, em particular, de sistemas de informação Web?
- Quais conflitos de código podem ser resolvidos de forma automatizada, semi-automatizada ou manual quando se está reconciliando LPS clonadas?
- Qual o impacto de tais conflitos de integração de código das LPS clonadas ao longo das suas camadas, assim como no núcleo e variabilidades das mesmas?

1.4 Objetivos Gerais e Específicos

O objetivo geral deste trabalho é a proposição e implementação de uma abordagem para evolução e reconciliação de linhas de produtos de software clonadas orientada a tarefas. Nesta abordagem, a caracterização das evoluções dos produtos são obtidas através da extração de informações a partir da mineração de informações de tarefas e do código-fonte, que são recuperadas em ferramentas de gestão de mudanças e repositório de código. A partir de tais informações, é possível obter o histórico de quais tarefas de desenvolvimento foram realizadas e que estruturas de código (classes, interfaces, métodos, atributos) foram modificadas para cada uma das tarefas. A abordagem então utiliza essas informações de evolução de cada LPS clonada para quantificar que tipos de conflitos léxicos, estruturais e semânticos podem ocorrer durante a integração do código de uma LPS clonada para outra, e quais deles poderiam (ou não) serem integrados de forma automatizada ou semi-automatizada.

Os objetivos específicos deste trabalho são:

- propor e implementar uma abordagem que permita a evolução e reconciliação de LPS clonadas implementadas na linguagem Java;
- implementar uma ferramenta de suporte à abordagem que permita a identificação de conflitos durante a reconciliação automática de LPS clonadas, de forma a facilitar a resolução automatizada, semi-automatizada ou manual de tais elementos;

- conduzir um estudo empírico de avaliação da abordagem e ferramenta proposta para o contexto de LPS de sistemas de informação Web com o objetivo de caracterizar quais tipos de conflitos ocorrem durante a evolução das mesmas;
- analisar o impacto dos conflitos de reconciliação de código de LPS clonadas nos diferentes tipos de tarefas, nas camadas/módulos da LPS e no seu núcleo e variabilidades;
- analisar e comparar o trabalho proposto com outros trabalhos de pesquisa conduzidos pela comunidade de engenharia de software e sistemas colaborativos.

1.5 Organização do Documento

Esta tese está estruturada da seguinte forma:

- O Capítulo 2 descreve a fundamentação teórica deste trabalho;

- O Capítulo 3 apresenta uma visão geral das linhas de produto de sistemas de informação Web investigadas nesta tese de doutorado, mostrando suas similaridades e variabilidades, e técnicas composicionais e anotativas utilizadas na implementação das suas variabilidades;

- O Capítulo 4 descreve em detalhes a abordagem proposta para análise e resolução de conflitos de LPS clonadas. Inicialmente, identifica-se cenários de evolução concomitantes para melhor contextualização da solução proposta. Em seguida, cada componente da abordagem é explicado individualmente assim como a forma como eles interagem para atender o objetivo proposto pela abordagem;

- O Capítulo 5 detalha o projeto e implementação da abordagem, descrevendo seus elementos arquiteturais, componentes de implementação e ilustrando a ferramenta de apoio desenvolvida no ambiente Eclipse;

- O Capítulo 6 apresenta o estudo empírico de caracterização de conflitos que ocorrem quando reconciliando LPS clonadas. Em particular, são analisadas quatro diferentes evoluções de LPS clonadas do SIG-UFRN;

- O Capítulo 7 apresenta os trabalhos relacionados;

- O Capítulo 8 apresenta as conclusões, limitações do trabalho e propostas futuras e finalmente;

- O Anexo A apresenta dados complementares do estudo empírico descrito no Capítulo 6.

2. Fundamentação Teórica

Este capítulo apresenta uma visão geral dos tópicos de fundamentação teórica necessários ao entendimento do desenvolvimento desta tese. A Seção 2.1 apresenta conceitos de linhas de produto de software (LPS). A Seção 2.2 discute sobre o gerenciamento de conflitos e *merge* de código. E por fim, a Seção 2.3 apresenta uma visão geral sobre a técnica da clonagem em linhas de produto de software.

2.1 Linhas de Produto de Software

O sonho da massificação do reuso de software é contemporânea ao início da engenharia de software. Diversas tentativas ou iniciativas de reuso foram feitas, mas inicialmente, com pouco sucesso. As propostas iniciais de reutilização eram, em geral, abordagens de pequena escala que promoviam a reutilização de código de forma localizada, mas ainda sem um foco no processo de desenvolvimento como um todo.

A proposta de se concentrar em um domínio específico como base para o desenvolvimento de artefatos reusáveis somente foi introduzido um pouco mais tarde (Neighbors, 1984). No entanto, os trabalhos neste contexto eram quase todos focados em desenvolvimento automatizado de software baseado em ferramentas de geração de código. Estes trabalhos focavam-se em linguagens de domínio específico, mas nunca escalando para o desenvolvimento em larga escala. Já o conceito de família de sistemas foi introduzido por Parnas (Parnas, 1976) na década de 70.

O conceito de linhas de produto de software foi introduzido no início da década de 90. Uma das primeiras contribuições foi a descrição do método *Feature-Oriented Domain Analisys* (FODA) por Kang et al (Kang et al., 1990). Na mesma época várias empresas começaram a tratar deste problema de forma mais sistemática. A Philips, por exemplo, introduziu um dos primeiros métodos (Linden &. Muller, 1995) na área e depois diversas outras empresas investiram na engenharia de linha de produtos, assim como a comunidade científica.

A diferença chave entre o processo de desenvolvimento de software tradicional e o de uma linha de produto de software é o foco de uma visão estratégica de um segmento de mercado, ao invés de gerar produtos individuais para atender um determinado contrato. Diferentes razões levam as empresas a adotar métodos, técnicas e práticas de engenharia de LPS, dentre as quais estão o custo e o tempo, pois há uma redução geral nos custos, uma diminuição no tempo de entrega do produto ao mercado e um aumento na qualidade do produto, bem como de sua confiabilidade. A diminuição dos custos e redução do tempo de entrega do produto ao mercado está fortemente correlacionada à abordagem de reuso em larga escala durante o desenvolvimento de software. Em oposição às abordagens tradicionais (Poulin,1997), o reuso proporcionado pela LPS pode representar 90% do software como um todo. A reutilização traz um melhor custo-benefício quando comparada com o desenvolvimento sob demanda. Dessa forma, tanto o custo de desenvolvimento quanto o tempo de entrega do produto podem ser reduzidos na engenharia de linha de produto. Infelizmente, este incremento não é gratuito, é necessário um investimento inicial maior do que a estratégia tradicional, porém, em muitos casos, é equivalente ao investimento de desenvolvimento de três produtos individuais (Linden et al., 2007), conforme ilustra a Figura 2-1. A partir de então, a estratégia de engenharia de linhas de produto de software pode torna-se economicamente mais vantajosa.

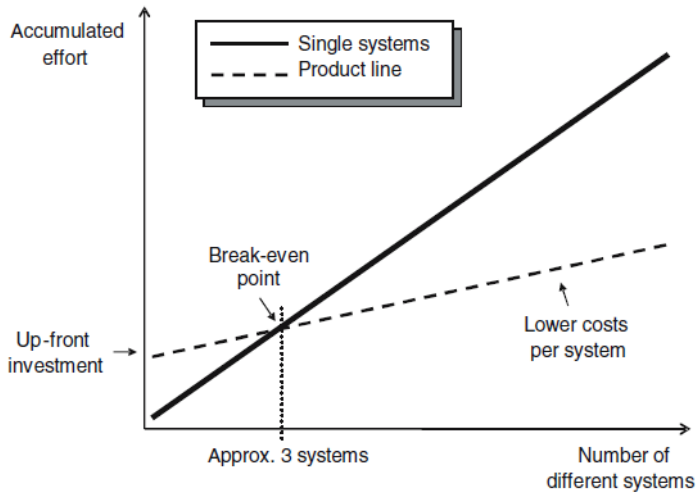


Figura 2-1 Economia da Engenharia de Linha de Produto de Software (Linden et al., 2007)

Métodos de engenharia de linhas de produto de software propõem uma distinção fundamental entre o desenvolvimento de software *para reuso* e o desenvolvimento de software *com reuso,* como mostrado na Figura 2-2. Na engenharia de domínio (desenvolvimento para reuso) a base é fornecida para o desenvolvimento de produtos individuais. Ao contrário de várias abordagens tradicionais de reuso focadas em artefatos de código, a infraestrutura da linha de produto engloba todos artefatos que são relevantes ao ciclo de desenvolvimento. Estes artefatos vão desde os requisitos à arquitetura até a implementação aos testes. Esta coleção de artefatos define a infraestrutura da LPS. Outra importante distinção do modelo tradicional é que os vários artefatos contém variabilidades explícitas. Por exemplo, a representação dos requisitos deve conter descrições explícitas das variabilidades que se aplicam somente aos subconjuntos dos produtos.

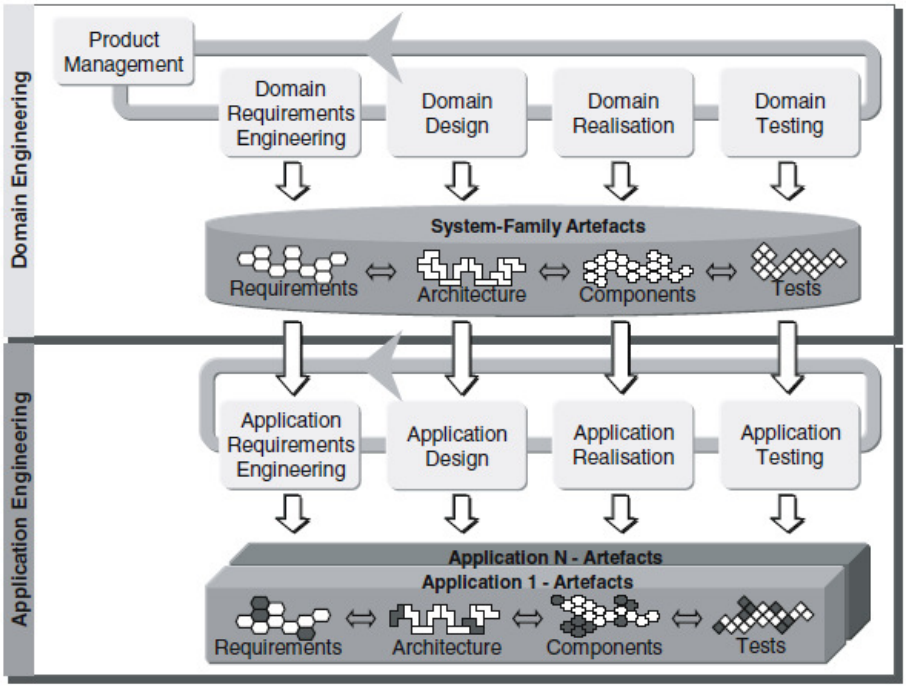


Figura 2-2 O modelo de dois ciclos da Engenharia de Linha de Produto (Linden et al, 2007)

Os artefatos individuais na infraestrutura da LPS são inter-relacionados assim como os artefatos do desenvolvimento tradicional de software. Por exemplo, a rastreabilidade é definida entre artefatos individuais, permitindo obter requisitos e identificar todas as implementações e casos de testes relacionados. A engenharia de aplicação (desenvolvimento com reuso) desenvolve o produto final usando como base a

infraestrutura da LPS. A infraestrutura define previamente a maioria das funcionalidades requeridas para um novo produto. As variabilidades explicitamente modeladas nesta infraestrutura fornecem a base para a derivação de produtos individuais. Basicamente, quando um novo produto é desenvolvido, ele é gerado a partir do núcleo de artefatos da LPS que definem as funcionalidades comuns de todos os produtos. Os requisitos são levantados e então categorizados em três tipos: (i) comum a todos os produtos, (ii) variáveis de acordo com o produto gerado ou (iii) específicos de um determinado produto. A partir desta análise, os vários artefatos de software (arquitetura, implementação, dentre outros) devem ser corretamente relacionados à categorias de requisitos para que possam ser instanciados de forma correta permitindo a derivação de um produto de software desejado.

Vários princípios são fundamentais para o sucesso de uma abordagem de engenharia de linha de produto de software. Os principais são descritos a seguir (Liden et al, 2007):

- **Gerenciamento de Variabilidades:** sistemas individuais são considerados como variações de um tema comum. A variabilidade é explícita e deve ser sistematicamente desenvolvida;

- **Centrado no Negócio:** a engenharia de linhas de produto de software deve estar conectada com a estratégia de longo prazo do negócio;

- **Centrado na Arquitetura:** a estrutura do software deve ser desenvolvida de forma a permitir obter vantagens das similaridades entre os sistemas individuais;

- **Abordagem de Ciclo de Vida Duplo:** Os sistemas individuais são desenvolvidos baseados em plataforma de software. Estes produtos – como também a plataforma – devem ser desenvolvidos e ter seus ciclos de vidas individuais.

2.1.1 Gerenciamento de Variabilidades

Um dos objetivos da engenharia de LPS é oferecer suporte a uma gama de produtos que devem atender clientes diferentes ou individuais, pertencendo a um mesmo segmento de mercado. Dessa forma, a variabilidade é um conceito chave na abordagem. Ao invés de entender um sistema individual como um todo, os métodos e técnicas de LPS buscam o entendimento da LPS, através das variações que existem entre seus sistemas individuais. Esta variabilidade deve ser definida, representada, explorada, desenvolvida, testada e evoluída – resumindo, devem ser gerenciadas de forma adequada.

Para realizar o gerenciamento de variabilidades em LPS é necessário distinguir uma característica do sistema em três principais tipos:

Similaridade: uma característica (funcional ou não funcional) que é comum a todos os produtos da linha. Chamamos essa característica de similaridade. Ela é então implementada como parte da plataforma da LPS;

Variabilidade: uma característica é comum para alguns produtos, mas não para todos. Ela deve ser explicitamente modelada como uma possível variabilidade e deve ser desenvolvida de forma que seja incorporada apenas aos produtos que tem esta característica;

Específica de Produto: A característica que só faz parte de um produto – pelo menos em um futuro previsível. Essas variabilidades normalmente não significam uma característica de mercado, mas a exigência de um cliente individual. Tais variabilidades não devem ser integradas a plataforma, porém devem ser capazes de fornecer o suporte a ela.

Durante o ciclo de vida da LPS uma variabilidade específica pode mudar de tipo. Por exemplo, uma característica específica de um produto pode tornar-se uma variabilidade. Por outro lado, uma característica comum pode tornar-se também uma variabilidade.

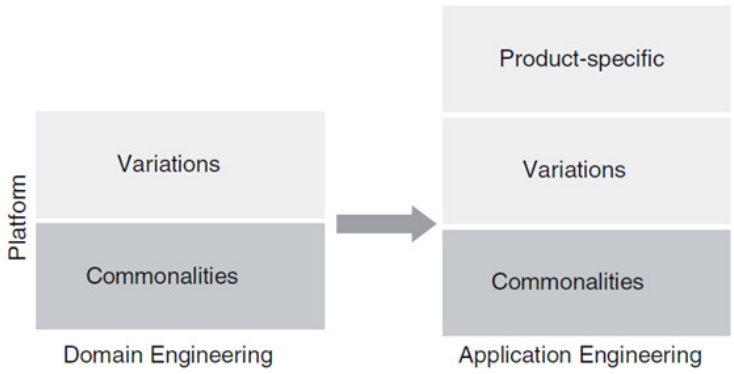


Figura 2-3 Relação entre os diferentes tipos de variabilidades (Linden et al., 2007)

Enquanto similaridades e variabilidades são manipuladas na engenharia de domínio, as partes específicas de produtos são manipuladas exclusivamente pela engenharia de aplicação. A Figura 2-3 ilustra tal cenário.

As características de uma LPS são categorizadas nos seguintes tipos:

- **Característica obrigatória:** característica presente em todos os produtos. Representam as similaridades;
- **Característica Opcional:** uma variabilidade que pode ser ativada ou não em um determinada produto, normalmente indicada por um valor lógico: ou está ou não está presente;
- **Característica Alternativa:** uma determinada variabilidade possui várias opções de escolha e uma delas pode ser selecionada;
- **Característica *OR*:** uma determinada variabilidade possui várias opções de escolha e uma ou mais podem ser selecionadas.

As características são modeladas através de um modelo de características (*Feature Model*) que as inter-relaciona através de uma árvore hierárquica com notações próprias para cada tipo de característica e suas dependências.

De acordo com Svahnberg (Svahnberg et al, 2001) variabilidade é a habilidade de mudar ou customizar um sistema. Aumentar a variabilidade de um sistema torna

mais fácil realizar mudanças. Alguns tipos de variabilidades já podem ser antecipadas, outras, só demonstram-se em estágios mais avançados do ciclo de vida do software.

Reusabilidade e flexibilidade tem sido tratadas como metas no desenvolvimento através de técnicas orientada a objetos, framework orientado a objetos e em LPS. Desta forma, estas técnicas nos permite atrasar certas decisões de projeto para pontos mais avançados no desenvolvimento. Com o uso da LPS a arquitetura do sistema é fixado antecipadamente mas os detalhes da implementação do produto podem ser retardadas até a fase da construção do produto. Estas decisões de projeto são chamadas de pontos de Variabilidade (Svahnberg et al, 2001).

.

2.1.2 Técnicas para Implementação de Variabilidades

Existem diversas técnicas para implementação de variabilidades em linhas de produto de software. Uma das classificações adotadas pela literatura (Kästner, 2010) as agrupa em abordagens composicionais e anotativas.

Na abordagem composicional, as características de uma LPS são implementadas separadamente em módulos (arquivos, classes, pacotes, plug-ins, dentre outros). Durante o processo de derivação de produtos estes módulos podem ser compostos em diferentes combinações. Técnicas de implementação consideradas composicional incluem: frameworks, aspectos, *mixin layers* e padrões de projetos. Esta abordagem utiliza técnicas mais sofisticadas e em geral, mais apropriada para implementar variabilidades de granularidade alta (Kästner, 2010).

Na abordagens anotativa o controle da escolha das variantes da LPS é feito através de fragmentos de códigos anotados. Um exemplo clássico desta técnica é a compilação condicional utilizada com pré-processadores de códigos fonte. A partir da seleção de características da configuração do produto, os fragmentos de códigos que implementam cada dessas características são selecionadas e inseridos no código fonte. Uma técnica similar é a da execução condicional (Santos et al., 2012), onde as variantes são configuradas através de parâmetros em banco de dados ou arquivos de configuração, e em tempo de execução decide-se qual fragmento de código será executado. Diferente da compilação condicional onde o código é inserido por um pre-

processamento, a execução condicional contém na base do código fonte todas as opções de variantes com a escolha realidade em tempo de execução.

Esta tese de doutorado investiga problemas relacionados a conflitos de código quando se está evoluindo LPS clonadas. Em particular, o trabalho explora e conduz estudos no contexto de LPS de sistemas de informação Web, que utilizam técnicas composicionais e anotativas na implementação de suas variabilidades. O Capítulo 3 apresenta as LPS web investigadas nessa tese, assim como as técnicas composicionais e anotativas usadas na implementação de suas variabilidades.

2.2 Gerenciamento de Conflitos de Merge de Código

A ocorrência de conflitos de código oriundos de manutenções paralelas realizadas por diferentes desenvolvedores de um mesmo produto é um desafio enfrentado na engenharia de software moderna. Tais conflitos trazem custos adicionais com a realização de *merges* de códigos incompatíveis ou que geram comportamentos indesejáveis ao sistema. Cada membro de uma equipe de desenvolvimento possui seu espaço de trabalho privado (*workspace*) para criar, modificar e evoluir o código do sistema e, posteriormente, publicá-lo em um repositório compartilhado.

No desenvolvimento de software em equipes o uso de sistemas de controle de versões - SCV (ou sistema de controle de revisões) é uma prática comum e recomendada, pois permite gerenciar os diversos artefatos através de publicações incrementais de versões (ou revisões) de arquivos, permitindo o gerenciamento e rastreabilidade de quem realizou as modificações no decorrer do tempo. O suporte a novas e independentes linhas de desenvolvimento, denominado *branches*, também é uma característica presenta na maioria dos SCV. Usando esse recurso os artefatos são copiados em uma nova estrutura mantendo-se uma vinculação temporal de quando essas cópias foram realizadas, permitindo então uma reconciliação futura, através de uma operação denominada de *merge*. A operação de *merge* tradicional utiliza o algoritmo de *diff* (MacKenzie et al., 1997) para realizar uma análise léxica dos arquivos que representam cópias de classes do sistema, e então identificar quais mudanças foram efetivamente realizadas. Neste cenário, comparando-se apenas textualmente, não há como identificar e diferenciar elementos estruturais (atributos ou métodos, por exemplo)

do código ou mesmo dependência entre eles. Quando dois artefatos são modificados de forma concomitante um conflito é apresentado no *merge* e, em geral, a sua resolução é manual e de responsabilidade da equipe de desenvolvimento.

A busca pela diminuição dos conflitos de códigos ocorridos durante o processo de desenvolvimento tem sido alvo de diversas pesquisas. Brun et al (Brun et al., 2013) desenvolveram uma abordagem especulativa para tentar antecipar os conflitos antes deles se manifestarem, prevenindo que desenvolvedores publiquem seus códigos em conflitos com os de outros. Guimarães & Silva (Guimarães & Silva, 2012) propõem uma abordagem de *merge* contínua, que analisa os códigos publicados e não publicados em espaços de trabalhos locais e indica os diferentes tipos de conflitos antes da publicação do código. Esta abordagem também busca prevenir os conflitos antes deles serem publicados pelos desenvolvedores. Wloka et al (Wloka et al., 2009) propuseram uma abordagem para analisar o *software* em desenvolvimento e identificar mudanças que são passíveis de publicação no repositório de código através de diversas políticas de publicação sem causar erros de testes. Para determinar se ocorrem ou não erros a abordagem extrai do modelo de mudanças as operações atômicas (adição, remoção e atualização de métodos e atributos) e os relaciona com o grafo de chamadas obtido através da execução dos testes unitários. Outros trabalhos relacionados são os que abordam o desenvolvimento consciente (*awareness development*), ou seja, aquele que permite um desenvolvedor entender as atividades que estão sendo realizadas por outros desenvolvedores, proporcionando um contexto para suas próprias atividades (Dourish & Bellotti, 1992). Neste contexto, ferramentas foram propostas para o desenvolvimento colaborativo com o objetivo de aumentar as informações do trabalho da equipe para membros individuais e assim evitar que conflitos de códigos possam ocorrer. Dentre estas ferramentas destacam-se a *Palantir* (Sarma, et al., 2012) e a *Syde* (Hattori & Lanza, 2010) que serão discutidas no capítulo 7 desta tese.

2.3 Clonagem em Linhas de Produto de Software

Considere a meta de desenvolvimento de um novo sistema que é muito similar à algum já feito anteriormente ou adaptando-o para um cenário de requisitos distintos. É possível usar um sistema existente, gerar uma cópia, modificar o que for necessário, e

então trata-lo como novo produto com uma trajetória própria de manutenção. Esta abordagem de reuso é denominada de *clone and own* (SEI, 2012).

A engenharia de linhas de produto de software se desenvolveu sob a ótica do reuso como uma disciplina planejada e previsível, desta forma, utilizar cópias de artefatos como técnica de reuso tem sido combatido com frequência. Fowler advoga: "Se você vir o mesmo código em mais de um lugar, pode ter certeza que seu software será melhor se você encontrar uma forma de unificá-los" (Godfrey, 2013). Johnson advoga: "Copiar e colar não é necessariamente ruim a curto prazo, se você está copiando código. Mas, ele é sempre ruim a longo prazo" (Godfrey, 2013). Apesar destas opiniões, verifica-se que a flexibilidade e rapidez do *time-to-market* gerada pela clonagem faz com que muitas empresas utilizem esta abordagem (Dubinsky et al., 2013).

A simplicidade da clonagem começa a diminuir quando a quantidade de clones vai aumentando e a gerência dos requisitos e modificações feita em várias versões de código torna-se um desafio. Mudanças feitas em um clone não são facilmente incorporadas em outros, tornando necessário a inclusão de um processo de gerenciamento dos clones. Uma primeira estratégia seria unir as características de cada clone em uma única cópia que unifica todos os comportamentos, estratégia denominada de *merge-refactoring* (Rubin & Chechik, 2012)(Mende et al.,2009). Esta abordagem é custosa e demanda muito tempo, não trazendo os benefícios imediatos tão valorizados por quem utiliza a clonagem. Uma segunda estratégia de gerenciamento é utilizar técnicas de engenharia de LPS para gerenciar as mudanças ocorridas nos clones, a denominada engenharia de LPS baseada em clones, dentre as quais podemos citar a proposta contida em Rubin (Rubin et al., 2013) e a abordagem desta tese. As abordagens propostas para gerenciamento de clones são detalhadas e comparadas com o trabalho proposto nesta tese na Seção 7.3 do Capítulo 7.

2.4 Sumário

Ao longo deste capítulo foram apresentados conceitos relacionados às linhas de produto de software relacionados com esta tese. O conceito de linha de produto de software, as suas vantagens estratégicas, o gerenciamento de variabilidades e as técnicas

de implementação destas características. As características são categorizadas como (i) comuns a todos os produtos, (ii) variáveis entre os produtos e (iii) específicas de cada produto. Os conflitos de software oriundos do desenvolvimento paralelo é um desafio enfrentado na engenharia de software moderna, as técnicas de resolução incluem abordagens específicas de resolução ou as realizadas diretamente utilizando os sistemas de controle de versão. Finalmente, a clonagem de linha de produtos foi apresentada como uma técnica de reutilização de LPS que apesar de historicamente combatida tem sido amplamente utilizada atualmente.

3 Linhas de Produto de Software de Sistemas de Informações Web

Este capítulo apresenta linhas de produto de software voltadas para o domínio de sistemas de informações Web explorados no contexto desta tese. A Seção 3.1 apresenta uma visão geral de linhas de produto de sistemas de informação Web desenvolvidas para a Universidade Federal do Rio Grande do Norte. A Seção 3.2 descreve as variabilidades existentes nas linhas de produto. A Seção 3.3 apresenta a arquitetura adotada por tais linhas de produto. A Seção 3.4 discorre sobre as duas técnicas utilizadas – composicional e anotativa – na modularização de variabilidades das linhas de produto. A Seção 3.5 analisa os impactos de utilização de tais técnicas no contexto de evolução de uma linha de produto de sistemas de informação Web. Finalmente, a Seção 3.6 sintetiza os assuntos abordados no capítulo.

3.1 Linhas de Produto para Sistemas de Informação Web

Os sistemas de informações web são largamente utilizados para criação de software corporativos pela sua flexibilidade de acesso e padronização tecnológica. As corporações cada vez mais dependem de seus sistemas para as atividades do dia e dia, o que faz com que o tempo de resposta às demandas de mudanças necessite ser cada vez menor. Estas mudanças podem ocorrer em vários cenários: mudanças de estratégias de negócios dentro da organização, mudanças de legislações ou outros fatores externos ou mesmo a utilização em outros cenários ou clientes.

A adoção da engenharia de linha de produto de software pode tornar os sistemas web mais aderentes aos cenários de evolução, através do uso de técnicas que permitem implementar variabilidades específicas demandadas por diferentes clientes de um dado sistema, facilitando sua adaptação e customização para diferentes cenários. Este trabalho aborda linhas de produto de sistemas de informação web, desenvolvidos originalmente pela UFRN, e que são atualmente evoluídos de forma independente por diferentes instituições nacionais. Em particular, o trabalho aborda duas destas linhas de

produto que recebem o nome de sistemas, mas oferecem diversas opções de variabilidades que podem ser customizadas por seus clientes, sendo eles: o SIGAA – Sistema Integrado de Gestão de Atividades Acadêmicas e o SIPAC – Sistema Integrado de Gestão de Patrimônio, Administração e Contratos.

O SIGAA foca na área acadêmica através de seus módulos de graduação, pós-graduação (stricto e lato-sensu), ensino técnico, ensino médio e infantil, submissão e controle de projetos e bolsistas de pesquisa, submissão e controle de ações de extensão, submissão e controle dos projetos de ensino (monitoria e inovações no ensino), registro e relatórios da produção acadêmica dos docentes, atividades de ensino a distância e um ambiente virtual de aprendizado denominado Turma Virtual, dentre outros. Disponibiliza também portais específicos para: reitoria, professores, alunos, tutores de ensino a distância, coordenações lato-sensu e stricto-sensu e comissões de avaliação institucional e docente (Lima & Ferreira, 2008).

O SIPAC é um sistema administrativo de Governo Eletrônico que informatiza a área orçamentária da instituição e as consequentes requisições que demandam este orçamento (Material, Passagens, Diárias, Suprimento de Fundos, Auxílio Financeiro, prestações de serviço pessoa física e jurídica, dentre outras). Possui funcionalidades para gestão de almoxarifados, controle patrimonial, compras e licitações, controle de atas e pedidos em atas de registros de preços, acompanhamento de entrega de empenhos (liquidação de despesas), controle de obras e manutenções de bens imóveis, aquisição de materiais informacionais, faturas de água e energia elétrica, controle dos contratos e convênios, fluxo de processos e documentos eletrônicos, registro e pagamento de bolsistas, acompanhamento das despesas com automóveis e combustíveis. O SIPAC também disponibiliza portais de informações especiais para setores administrativos, dirigentes da instituição, para auditoria e controle interno e para a fundação de apoio à pesquisa (Lima & Ferreira,2008).

Os sistemas SIPAC e SIGAA possuem uma alta quantidade de funcionalidades e atendem diversas áreas dentro de uma instituição pública e/ou de ensino. Assim, o uso de técnicas de linhas de produto de software são importantes para permitir a estes sistemas uma rápida resposta às mudanças de negócio. Estes sistemas foram iniciados em 2004 e, a partir do ano de 2008 começaram a ser utilizados por outras instituições, o que tornou as mudanças de contexto de negócio ainda mais frequentes e complexas. A

partir deste contexto, estes sistemas começaram a ser flexibilizados, para possibilitar a implantação de determinadas customizações específicas (variabilidades), dando origem, cada um deles, a uma linha de produto de sistemas de informação Web. A Tabela 3-1 enumeras algumas métricas destas linhas de produto:

Linha de Produto	**Linhas de Código (LOC)**	**Total de Classes**	**Páginas WEB**	**Tabelas de Dados**	**Funcionalidades**
SIGAA	1,5 milhão	4.900	4.000	1.150	1.850
SIPAC	1,2 milhão	4.356	4.000	1.200	1.780

Tabela 3-1 Métricas das Linhas de Produto Estudadas (LIMA,2012)

3.2 Variabilidades das Linhas de Produto SIGAA e SIPAC

A introdução de variabilidades nos sistemas SIG, nome utilizado para referenciar a família dos sistemas em estudo, é importante para diminuir os cenários de conflitos de código entre as várias versões da linha de produto, assim como, permitir flexibilidade de variações de comportamentos no produto para diferentes contextos das organizações.

A LPS SIGAA possui como principal foco o ensino superior através da sua indivisibilidade entre ensino, pesquisa e extensão. A constituição federal, no seu artigo 207, outorga direitos de autonomia didático-pedagógica para as universidades, permitindo assim que cada uma defina seus próprios modelos e regras, respeitando apenas fundamentos básicos previstos na lei 9.394, a lei de fundamentos e bases da educação (LDB). Neste cenário, as variabilidades introduzidas na LPS SIGAA possuem uma importância relevante considerando o cenário quase certo de mudança de regras entre as diversas universidades.

Um exemplo dessa variabilidade é o processo de fechamento ou consolidação de turmas. Neste processo, após a digitação das notas, o sistema aplica as regras de negócio para atualizar as situações dos discentes para aprovado, reprovado, em recuperação, dentre outros. Neste caso, o SIGAA possui uma variabilidade mandatória,

que permite a instituição usuária utilizar a implementação *default* (a usada na UFRN) ou definir o seu mecanismo de consolidação sem gerar conflitos de código. A Tabela 3-2 enumera as variabilidades do sistema SIGAA, levantadas no escopo deste trabalho:

Código	**Módulo**	**Descrição**	**Tipo**
base-01	Geral	Permite adicionar/remover comportamentos no processamento de vínculos através de uma cadeia de processamentos.	OR-Feature
ensino-01	Ensino	Estratégia para geração de matrículas dos discentes.	Alternativa
ensino-02	Ensino	Estratégia para a consolidação de turmas	Alternativa
ensino-03	Ensino	Estratégia para cálculo dos prazos máximos de cursos	Alternativa
grad-01	Graduação	Cálculo dos índices acadêmicos de discentes	OR-*Feature*
grad-02	Graduação	Cálculo do perfil inicial de estudantes de graduação	Alternativa
grad-03	Graduação	Interface para regras de convocação de candidatos em processos seletivos de graduação	Alternativa
grad-04	Pesquisa	Interface para obter classificação dos alunos em turmas visando o processamento	Alternativa
proj-01	Pesquisa	Interface para generalização do comportamento dos cálculos de pesquisa	Alternativa
proj-02	Projetos	Estratégia para avaliação de projetos acadêmicos (ensino, pesquisa e extensão)	Alternativa
proj-03	Projetos	Algoritmo padrão para distribuição de projetos para avaliadores.	Alternativa

bib-01	Biblioteca	Estratégia para obter os usuários a partir dos seus vínculos no sistema e de acordo com as regras do sistema de bibliotecas da instituição	Alternativa
bib-02	Biblioteca	Interface padrão para lógica de punição de atrasos de empréstimos	Alternativa
bib-03	Biblioteca	Interface padrão para implementações de buscas textuais	Alternativa
pi-01	Produção intelectual	Interface para geração de atividades ou produções intelectuais para relatórios de produções	Alternativa
ec-01	Avaliação Docente	Habilitação de funcionalidade da avaliação docente.	Opcional
ec-02	Biblioteca	Habilitação da funcionalidade de Reservas na biblioteca.	Opcional
ec-03	Biblioteca	Habilita as funcionalidades de suspensão no sistema de bibliotecas.	Opcional
ec-04	Biblioteca	Habilita as funcionalidades de multa no sistema de bibliotecas.	Opcional
ec-05	EAD	Habilita o trancamento de componentes curriculares no EAD.	Opcional
ec-06	EAD	Habilitação de funcionalidade de tutoria de EAD.	Opcional
ec-07	Graduação	Habilita a coordenação a realizar matricula de turmas de férias.	Opcional
ec-08	EAD	Permite aluno EAD fazer matrícula *online*.	Opcional
ec-09	Lato Sensu	Restringe ao professor coordenador a submissão de proposta de lato-sensu.	Opcional

ec-10	Stricto-Sensu	Permite programas de pós-graduação alterar a estrutura curricular dos cursos.	Opcional
ec-11	Stricto-Sensu	Permite coordenação alterar componentes curriculares.	Opcional

Tabela 3-2 Variabilidades do Sistema SIGAA

A LPS SIPAC possui uma base normativa melhor definida uma vez que a administração pública deve seguir os mesmos normativos e legislações. No entanto, isso não significa dizer que não existam mudanças, pois as leis definem pontos gerais e não abordam especificidades e fluxos internos. Por exemplo, a lei 8.666 (a lei de licitações e contratos) estipula as normas gerais da licitação, mas não impõe regras internas de como se dará a captação das demandas de compras dos diversos departamentos de uma instituição. Uma parcela das regras de negócios implementadas no SIPAC são definidas por fatores externos (legislações federais) e outra parcela é definida por normativos internos (regras definidas pela própria instituição). Assim, tornam-se também importantes as variabilidades neste sistema.

Um exemplo de variabilidade ocorre no módulo de patrimônio móvel. É preciso que toda instituição pública utilize um número de tombamento para identificar seus equipamentos, porém a forma desta numeração (sequencial geral, sequencial por ano, letras e números) é uma opção de cada instituição e variantes da LPS. Este comportamento é uma variabilidade do sistema SIPAC. A Tabela 3-3 enumera essas variabilidades identificadas do SIPAC:

Código	**Módulo**	**Descrição**	**Tipo**
alm-01	Almoxarifado	Controla os tipos de materiais que serão passíveis de utilização no almoxarifado.	*OR-Feature*
alm-02	Almoxarifado	Habilita o cadastro de tipos de saídas de materiais no módulo.	Opcional
cont-01	Contratos	Restringe o atendimento das requisições por unidade gestora.	Alternativa

cont-02	Contratos	Desativa os contratos acadêmicos.	Opcional
cont-03	Contratos	Controla se a aplicação permitirá editar livros de ocorrências de contratos.	Opcional
cont-04	Contratos	Habilita o fiscal a visualizar todos os dados do contrato.	Opcional
pat-01	Patrimônio	Estratégia para geração de número de tombo.	Alternativa
pat-02	Patrimônio	Permite escolher entre modelos de guia de movimentações de bens existentes.	Alternativa
pat-03	Patrimônio	Permite escolher o tipo de documento que será usado para o tombamento.	Alternativa
pat-04	Patrimônio	Habilita as funcionalidades de patrimônio par fundações de apoio.	Opcional
pat-05	Patrimônio	Habilita o controle de patrimônio por localidade.	Opcional
pat-06	Patrimônio	Configura a validação eletrônica de bens.	Opcional
pat-07	Patrimônio	Configura regras de tombamento de notas fiscais.	Alternativa
pat-08	Patrimônio	Habilita a edição de valores residuais de bens.	Opcional
prot-01	Protocolo	Habilita a funcionalidade de etiquetas de protocolo.	Opcional
prot-02	Protocolo	Configura o tipo de classificação de processos.	Alternativa
prot-03	Protocolo	Configura a utilização da classificação CONARQ.	Opcional
prot-04	Protocolo	Define regras de alteração de processos tramitados.	Alternativa

req-01	Requisições	Controla o bloqueio de requisições na ausência da prestação de constas da passagem.	Opcional
req-02	Requisições	Lista de processamentos em cadeia de bloqueio de requisições.	*OR-Feature*
fat-01	Faturas	Especifica o tipo de leitor de arquivo de faturas de concessionárias.	Alternativa
transp-01	Transportes	Especifica o interpretador de arquivo de abastecimentos de veículos.	Alternativa

Tabela 3-3 Variabilidades do Sistema SIPAC

3.3 Arquitetura de Linhas de Produto para Sistemas de Informação Web SIG

A arquitetura dos sistemas SIG da UFRN foi desenvolvida de 2002 a 2006 sob a ótica das melhores práticas de sistemas multicamadas utilizando a tecnologia *Java Enterprise Edition* (JEE). Em 2007, tal arquitetura foi modificada para buscar trazer melhorias de desempenho (Lima, 2007). No período de 2008 a 2011, ela incorporou boas práticas de *frameworks* baseados em POJO (*Plain Old Java Objects*) e inversão de controle (Fowler, 2012). Atualmente, esta arquitetura oferece suporte a quatro diferentes LPS, entre elas o SIGAA e o SIPAC, que juntas agregam mais de cinco milhões de linhas de código.

A arquitetura dos sistemas SIG segue o padrão arquitetural em camadas (Buschmann et al., 1996). Os artefatos de implementação são organizados e estruturados em quatro camadas principais:

1. **Camada de apresentação *(Presentation Layer):*** Trata dos aspectos de interação com o usuário (*controller*), através de um conjunto de classes *Managed Beans* do *framework Java Server Faces* (JSF), e a captação e exibição das informações (*view*) usando páginas *Java Server Pages* (JSP) codificadas com bibliotecas de *tags* JSF. O padrão *Model-View-Controller* (MVC) (Buschmann et al., 1996) é utilizado para organizar a camada de apresentação em

visualizações e controladores e reger sua comunicação com a camada de serviços;

2. **Camada de serviços ou negócio (*Service Layer*)**: Define os serviços oferecidos pela aplicação e delimita as transações a serem executadas pelo sistema através da sua implementação com *Enterprise Java Beans* (EJB). Nesta camada, o padrão de projeto EJB *Command* (Malks et al., 2002) foi aplicado para criar uma fachada de ativação das classes de lógica de negócio através de objetos comando e classes processadoras, responsáveis pela execução da lógica de negócios;

3. **Camada de Modelo de Domínio (*Domain Model*):** Contém as classes de entidade do sistema representando o domínio de negócio. Esta camada deve ser autocontida e sem dependências externas. Apenas anotações de persistência (*Java Persistence API*) são permitidas nesta camada;

4. **Camada de acesso a dados (*Data Access Layer*)**: Mantém as classes de acesso ao banco de dados (DAO – *Data Access Object*), que utilizam um *framework* de Mapeamento Objeto Relacional (MOR), para atualizar e recuperar informações do banco de dados. O *framework Hibernate* é usado para realizar o mapeamento dos objetos para o banco de dados e vice-versa.

A Figura 3-1 ilustra o diagrama de relacionamento de dependências entre estas camadas. A camada de apresentação, subdividida entre os seus elementos de interface (páginas JSP) e os elemento de controle (*Managed Beans*), depende basicamente de todas as camadas inferiores (serviço, acesso a dados e domínio). Observe-se que apenas os controladores devem depender da camada de serviço, sendo vedada essa dependência direta nos elementos de visualização (padrão MVC). A camada de serviço deve depender somente da camada de domínio e acesso a dados. A camada de acesso a dados depende apenas da camada de domínio. Já a camada de domínio deve ser autocontida, sem nenhuma dependência de outras camadas.

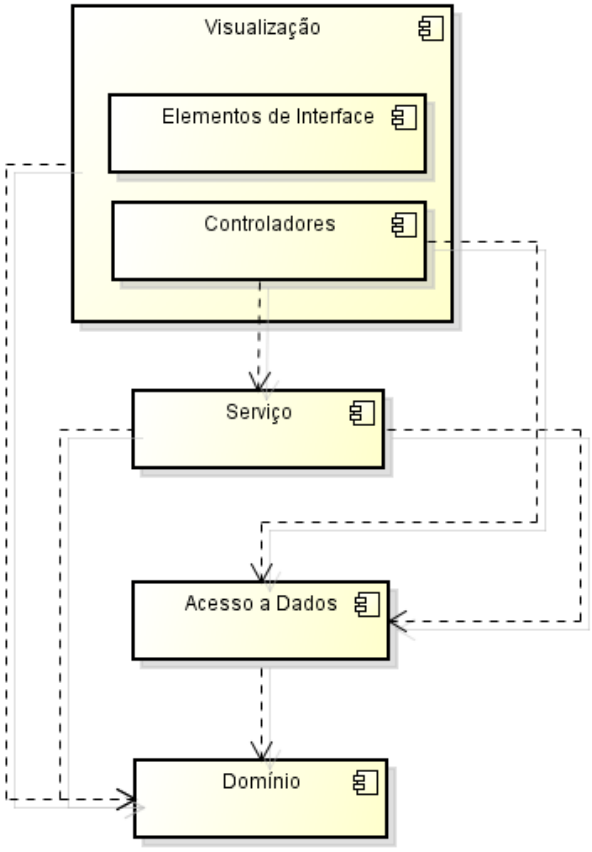


Figura 3-1 Diagrama de dependências na arquitetura

As variabilidades podem estar localizadas em uma determinada camada, como na camada de serviço, como também podem ter sua implementação distribuída em mais de uma camada. A ativação de uma determinada funcionalidade, por exemplo, tem impacto tanto na camada de visualização como na camada de serviço, já a criação de um ponto de extensão (*plugin*) para um comportamento de regra de negócio está localizada especificamente na camada de serviço.

Considere o exemplo de consolidação de turmas citado anteriormente, neste caso, a *feature* ***Estratégias de Consolidação de Turmas*** define um grupo de possíveis variantes suportadas para realizar a consolidação das turmas, sendo elas: (i) Avaliação por Competência, (ii) Educação a Distância, (iii) Graduação, (iv) Lato Sensu, (v) Médio, (vi) Metrópole Digital, (vii) Técnico de Música, (viii) Residência Médica; e (ix) Stricto Sensu. Cada uma das estratégias representa diferentes regras de fechamento de turmas por níveis de ensino diferentes (graduação, lato, stricto, dentre outros) ou por cursos com características peculiares, conforme ilustrado na Figura 3-2.

Esta variabilidade foi implementada através do uso integrado dos padrões *Strategy* e *Template Method* (Gamma et al., 1994) onde o algoritmo padrão para consolidação de turmas foi generalizado através da classe `AbstractEstrategiaConsolidacao` e é concretizado pela escolha de uma das diferentes implementações disponíveis. A classe da camada de serviço responsável pela regra de negócio de consolidação, `ProcessadorConsolidacaoTurma`, se utiliza do padrão *Factory Method* para decidir qual estratégia concreta utilizar.

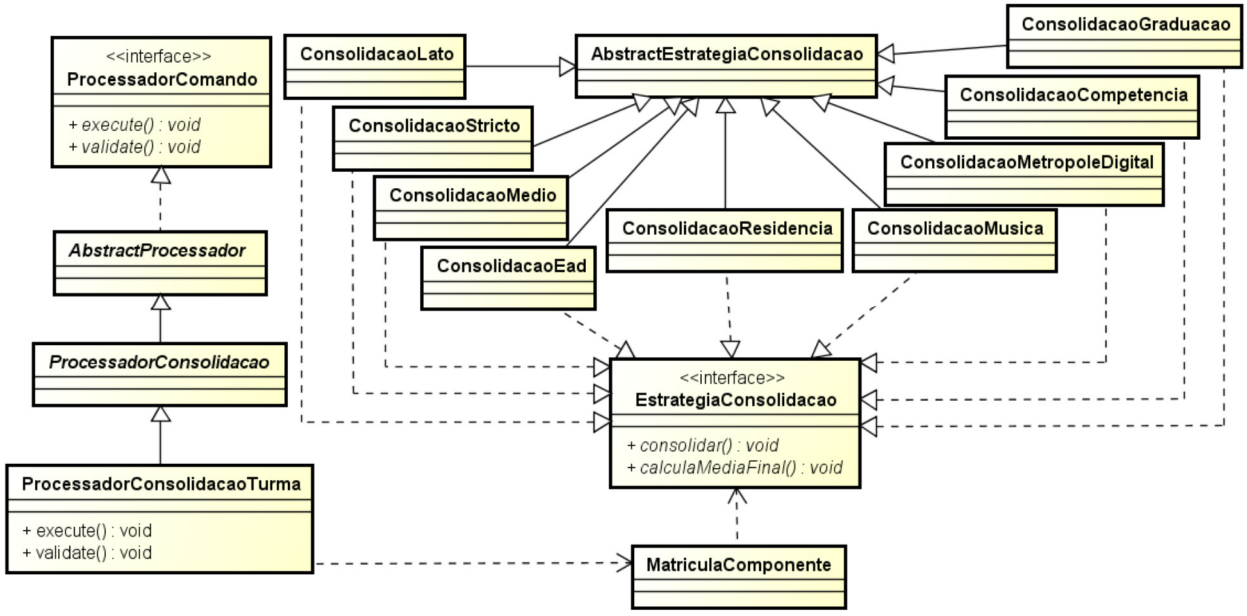


Figura 3-2 Variabilidade de Consolidação de Turmas (Pereira et al, 2010)

Outro exemplo de variabilidade é a leitura de arquivos de telefonia de operadoras do módulo de faturas do sistema SIPAC. As instituições usuárias podem possuir diferentes contratados para prestação de serviços de telefonia fixa e móvel, devendo o sistema estar apto a tratar os diferentes formatos dos arquivos disponibilizados (Claro, Embratel, Oi, Tim, dentre outras). A Figura 3-3 ilustra esta variabilidade.

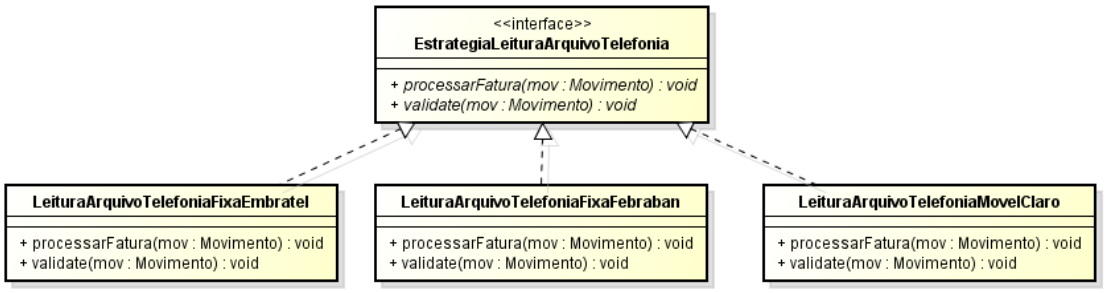


Figura 3-3 Estratégia de Leitura de Arquivos de Telefonia

Uma das variantes deve ser escolhida na configuração de um produto e outras podem ser criadas com baixo impacto de conflitos de forma a agregar as opções disponíveis na linha de produto. A implementação das variabilidades pode ocorrer através do uso de diversas técnicas utilizadas pela indústria de desenvolvimento de software. A seção a seguir aborda estas técnicas de implementação de variabilidades.

3.4 Técnicas para Implementação de Variabilidades

A adoção de uma arquitetura de software com técnicas de inversão de controle, flexibilidade de modularização e com camadas bem definidas oferece uma maior robustez para a implementação de novas funcionalidades e evolução das LPS. No entanto, ela não é suficiente para atender todas as necessidades de desenvolvimento de variabilidades na LPS.

As técnicas de modularização de variabilidades utilizadas nas LPS SIGAA e SIPAC podem ser divididas em duas categorias:

(i) **técnica composicional:** utiliza padrões de projetos orientados a objetos para implementar variabilidade de granularidade grossa;

(ii) **técnicas anotativa:** utiliza uma técnica, denominada *Execução Condicional*, que simula, em tempo de execução, o que a técnica de compilação condicional realiza durante o pré-processamento de um programa que será compilado.

3.4.1 Técnicas Composicionais

A utilização da técnica de padrões de projetos se iniciou com a clássica obra *Design Pattern: elements of reusable object-oriented Software* (Gamma et al., 1994), onde diversas boas práticas para solucionar problemas bem conhecidos de orientação a objetos foram catalogadas. Com o advento das plataformas corporativas de desenvolvimento, principalmente a *Java 2 Enteprise Edition,* diversos padrões de projeto específicos para implementação de arquitetura de sistemas corporativos foram propostos, tais como, os apresentados em *Core J2EE Patterns* (Malks et al., 2001) e, recentemente, em *Patterns of Enterprise Applications* (Fowler, 2012). Os padrões corporativos representam boas práticas na modelagem de problemas corporativos, tais como: escalabilidade de aplicações, padrões de integração, padrões arquiteturais, mecanismos de segurança, dentre outros.

Para a implementação de variabilidades em LPS, os padrões de projeto apresentados em (Gamma et al, 1994) ainda continuam sendo uma das mais relevantes contribuições na área. Por isso, serão detalhadas a seguir, com alguns exemplos de sua utilização. Os principais padrões utilizados para modelagem de variabilidades das LPS SIGAA e SIPAC são:

Nome do Padrão	**Utilização**
Factory Method	Permite que objetos sejam criados a partir de valores passados para métodos, instanciando assim o objeto correto que implementa um determinado comportamento padrão.
Chain of Responsibility	Divide um conjunto de responsabilidades entre diferentes objetos conectados, de forma que, se um objeto não souber responder a uma requisição, passa a função para o próximo objeto da cadeia.
Strategy	Define uma família de algoritmos, os encapsula e permite que eles sejam intercambiáveis.
Template Method	Define o esqueleto de um algoritmo e permite que alguns de seus passos sejam sobrescritos em classes filhas.

3.4.1.1 Factory Method

O padrão *Factory Method* é amplamente utilizado para a escolha de objetos apropriados de acordo com a configuração do produto da LPS. Para exemplificar seu uso, considere a variabilidade pat-01 (Tabela 3-3) sobre a geração de número de tombos. O sistema SIPAC define três variantes para esta característica: (i) Numeração Sequencial, (ii) Sequencial por ano e (iii) Sequencial por Unidade Gestora. A interface `EstrategiaGeracaoTombo` contém métodos relacionados com a geração do número do tombo. A classe `EstrategiaTomboFactory` contém o método `getInstance(String parameter)` que recupera o objeto correto a partir do parâmetro repassado.

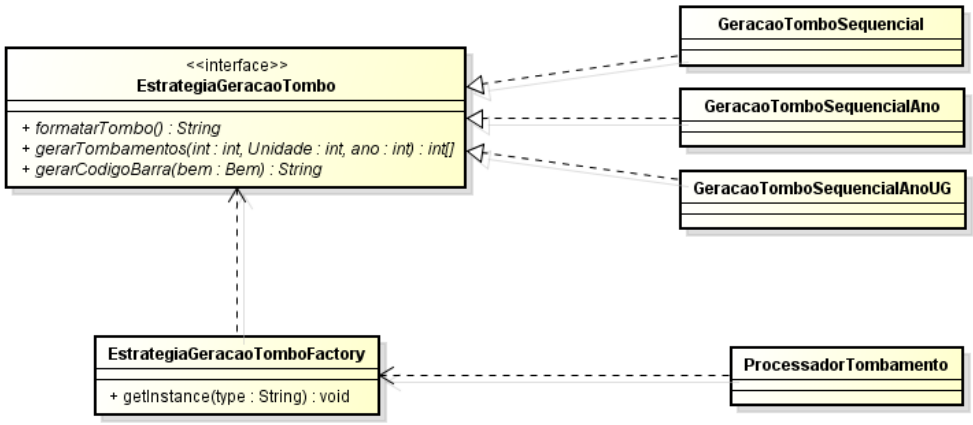


Figura 3-4 Fábrica para geração da estratégia de geração de números de tombamentos

A classe `ProcessadorTombamento` utiliza a fábrica de objetos de geração de tombos através do seguinte trecho de código:

```
EstrategiaGeracaoTombo geracaoTombo =
EstrategiaGeracaoTomboFactory.getInstance(ParametrosPatrimonio.ESTRATE
GIA_UTILIZADA);
```

3.4.1.2 Chain Of Responsibility

O padrão *Chain of Responsability* é aplicado nas LPS para implementar algumas das variabilidades do tipo *or-feature*. Este tipo de *feature* é um caso particular de *feature* alternativa, que permite a seleção de não apenas uma das características

pertencentes ao grupo alternativo de características, mas também a seleção de mais de uma característica desse grupo, ou até mesmo nenhuma delas.

No contexto das LPS dos sistemas SIG, exemplos de *or-feature* são os processamentos dos vínculos e permissões dos usuários. Estes processamentos podem ser compostos por diversas etapas e essas etapas podem variar de uma instituição para outra, ou dentro da mesma instituição, dependendo das mudanças de negócio. Através da aplicação do padrão *Chain of Responsability* é possível executar uma cadeia de classes que implementam um determinado processamento. As classes são executadas em uma ordem pré-definida, e ao término da execução de uma, inicia-se o processamento da classe seguinte. A Figura 3-5 pode-se verificar a classe base de execução (`AcessoMenu`) que cria a cadeira de execução através do método *createChain.* A adição e retirada de processamento na cadeia pode ser realizado através da criação ou remoção de novas classes na cadeira de execução.

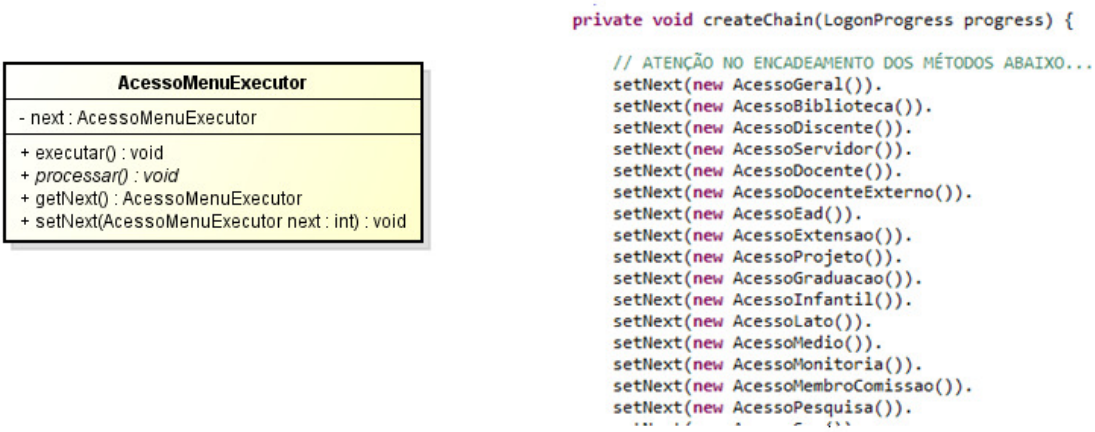


Figura 3-5 Criação da cadeira de processamento

3.4.1.3 Strategy

O padrão de projeto *Strategy* é um dos mais utilizados para implementação de variabilidades de granularidade alta em características alternativas. O padrão define uma família de algoritmos que varia independentemente do cliente que o usa.

Um exemplo de utilização do *Strategy* no contextos da LPS dos sistemas SIG é a autenticação de usuários. Para implementar este comportamento definiu-se uma interface padronizada com o algoritmo de autenticação (*interface* `EstrategiaAutenticacao`) e várias implementações concretas, que podem ser: (i)

com banco de dados; (ii) com LDAP (*Lightweight Directory Access Protocol*); e (iii) com banco de dados e LDAP. A escolha da variante é feita através do parâmetro `ESTRATEGIA_AUTENTICACAO`.

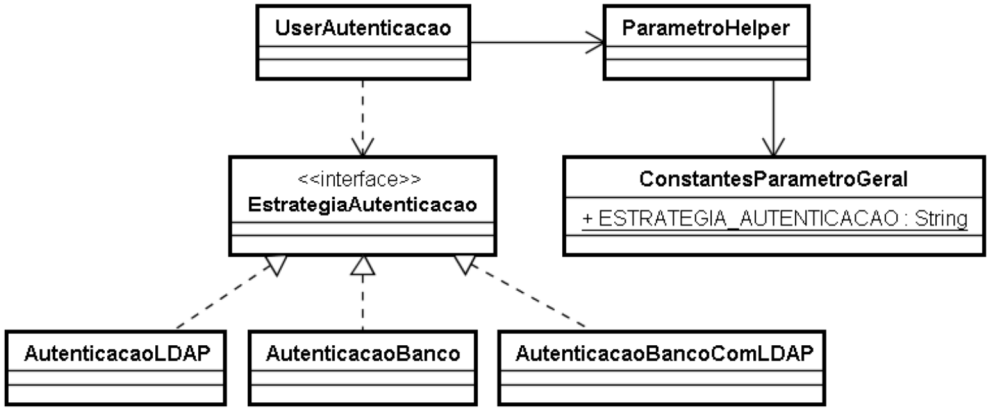


Figura 3-6 Uso do padrão de projeto Strategy na autenticação de usuários

O uso do padrão *Strategy* em combinação com o *Factory Method* fornece uma poderosa ferramenta para implementação de variabilidades. O *Strategy* fornece o mecanismo para generalização de comportamentos variáveis e o *Factory Method* uma forma eficaz da escolha da variante escolhida na *feature* alternativa.

3.4.1.4 Template Method

O padrão de projeto *Template Method* define um esqueleto de algoritmo de uma determinada operação, delegando alguns dos passos para subclasses. Ele delega certos passos do algoritmo para processamento da subclasse sem mudar a estrutura do algoritmo e sem gerar conflitos de código.

Um exemplo do uso do *Template Method* é a importação de produções da plataforma Lattes através de arquivo XML. O algoritmo de processamento possui um esqueleto padrão com determinadas variações por cada tipo de produção (participação em bancas, capítulo de livros, apresentações de trabalhos, dentre outras), pois, cada um possui dados específicos para serem tratados. A Figura 3-7 ilustra este cenário: a classe `ImportaProducaoLattesXML` possui o método padrão de processamento (método `processar`) e um método abstrato denominado `statElement`, que é usado para processamento do elemento XML. Cada tipo de produção possui uma implementação

concreta do `startElement` para tratamento correto das informações do tipo da produção. Todo o fluxo do algoritmo é invocado através da classe `ImportacaoProducaoLattesXML`, invocando as subclasses corretas durante a execução do processo.

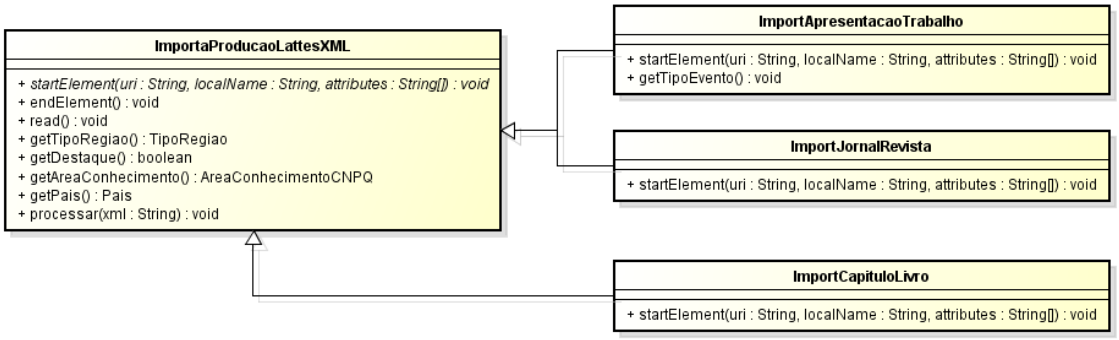


Figura 3-7 *Template Method* no processamento de produção do Lattes

3.4.2 Técnicas Anotativas

As técnicas anotativas atendem as variabilidades de granularidade fina e estão espalhadas no código fonte, normalmente através de anotações. A forma mais comum é a compilação condicional através das diretivas `#ifdef` e `#endif` através de pre-processadores de códigos para compilação. Exceto em poucas linguagens, tais como C#, D, Adobe Flex, anotações condicionais não são partes nativas da linguagem e sim adicionadas por ferramentas de mais alto nível. É possível utilizar compilação condicional em Java através de ferramentas externas como o *Antenna*[1] e *Munge*[2], dentre outras listadas em Kästner (Kästner, 2010). Um exemplo de uso é ilustrado na Figura 3-8.

[1] http://antenna.sf.net
[2] http://weblogs.java.net/blog/tball/archive/munge/doc/Munge.html

```
1  class Stack {
2    void push(Object o
3  #ifdef SYNC
4    , Transaction txn
5  #endif
6    ) {
7      if (o==null
8  #ifdef SYNC
9        || txn==null
10 #endif
11       ) return;
12 #ifdef SYNC
13     Lock l=txn.lock(o);
14 #endif
15     elementData[size++] = o;
16 #ifdef SYNC
17     l.unlock();
18 #endif
19     fireStackChanged();
20   }
21 }
```

Figura 3-8 Estrutura de Compilação Condicional em Java (Kästner, 2010)

O uso da compilação condicional, apesar de possível, não é usual na linguagem Java, conduzindo também a dificuldades na depuração em tempo de desenvolvimento, tornando o código de difícil entendimento, limitando-se a parâmetros booleanos e também propiciando a geração de produtos com erros de compilação (Kästner & Apel, 2009).

Outra técnica possível de ser aplicada é a execução condicional (Santos et al., 2012). Nesta técnica, as variabilidades são mapeadas através de parâmetros de configuração, provendo mecanismos para, a partir dos valores desses parâmetros, tomar decisões de qual variabilidade será efetuada em tempo de execução. Os parâmetros são armazenados em um meio persistente apropriado para facilitar o seu acesso, em geral, um banco de dados. Um componente se responsabiliza pela recuperação dos valores dos parâmetros de forma eficiente, retornado seus valores para as classes que irão executar o código da variabilidade que executarão as condições de execução a partir dos valores retornados. Na execução condicional todo o código está presente, sendo as partes do código ativadas a partir dos valores dos parâmetros, desta forma, é possível controlar melhor a depuração de código.

A Figura 3-9 ilustra a estrutura da execução condicional. As classes *Target* são componentes ou trechos de código onde ocorre a variabilidade, que podem ser utilizadas por outras camadas da aplicação, como a da apresentação, por exemplo. O código de implementação da variabilidade representa o trecho de código de granularidade fina que possui os testes da condição de execução. O configurador de

execução é um elemento central (Fachada) para recuperação dos parâmetros em tempo de execução. O repositório de parâmetros representa o mecanismo de armazenamento, que pode ser um banco de dados ou arquivos de propriedades, preferencialmente utiliza-se um banco de dados.

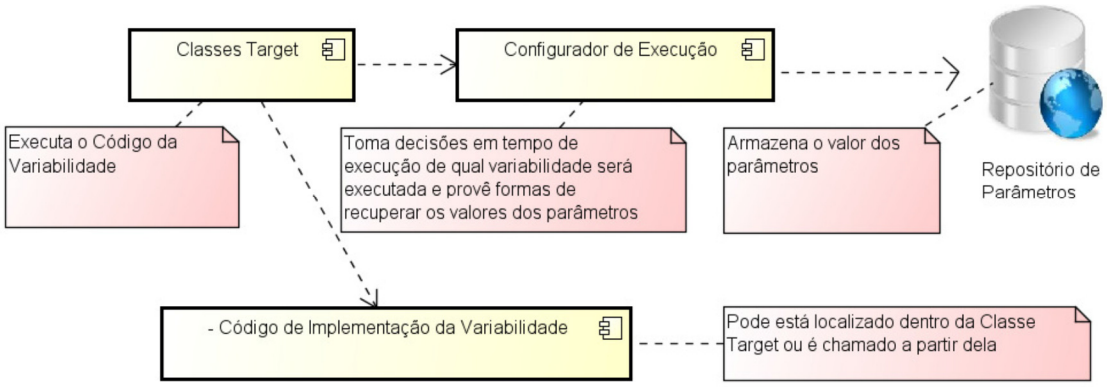


Figura 3-9 Estrutura da Execução Condicional (Santos et al., 2012)

Um exemplo de utilização da variabilidade de execução condicional é o tipo de documento a ser utilizado para os termos de tombamento no patrimônio. A instituição pode utilizar Termo de Responsabilidade (valor = `TipoDocumentoTombamento.TERMO_RESPONSABILIDADE`), ou então Termos de Transferência (valor = `TipoDocumentoTombamento.TERMO_TRANSFERENCIA`). A Figura 3-9 ilustra a execução do código de transferência.

```
    int        tipoDocumento        =        ParametroHelper.getInstance().getParametro(
ParametrosPatrimonio.TIPO_DOCUMENTO_TERMO_TOMBAMENTO );

     if ( tipoDocumento ==  TipoDocumentoTombamento.TERMO_RESPONSABILIDADE) {
        // trecho de código para geração do tombamento no termo de responsabilidade
    } else if (tipoDocumento ==  TipoDocumentoTombamento.TERMO_TRANSFERENCIA ) {
        // trecho de código para geração do tombamento no termo de responsabilidade
    }
```

Figura 3-10 Exemplo de Execução condicional

A execução condicional permite a escolha ou ativação de determinados comportamentos da aplicação de acordo com a configuração do produto. Ela serve para implementar características opcionais e alternativas na LPS de variabilidade fina.

3.4.2.1 Outras formas de parametrização

A técnica de parametrização de valores de negócio permite controlar fluxos do sistema, relacionando-os aos valores específicos do cliente que são recuperados através do configurador de execução (Figura 3-9). A Figura 3-11 ilustra o exemplo do tempo máximo para abertura do processo de homologação do processo de conclusão de uma pós-graduação após a defesa. Neste caso, tais valores variam entre as diversas universidades através dos seus normativos internos. Assim, a parametrização deste valor permite que a configuração de um determinado produto da linha seja feita de acordo com as regras locais, sem gerar conflitos de código.

```
// Obtém a instância de ParametroHelper
ParametroHelper helper = ParametroHelper.getInstance();

// Pega prazo máximo e calcula num. meses desde a defesa
int tempoMaximo = helper.getParametroInt(
        ConstantesParametro.TEMPO_MAXIMO_HOMOLOGACAO_STRICTO);
int meses = UFRNUtils.diferencaMeses(banca.getData(), new Date();

// Se o aluno extrapolou o limite de tempo, não deixar homologar
if (meses > tempoMaximo) {
   throw new NegocioException(...);
}
```

Figura 3-11 Parametrização de Valores de Negócio

Uma variação da técnica acima pode ser aplicada para recuperar parâmetros com valores múltiplos ou mesmo uma entidade de configuração. A Figura 3-12 ilustra um exemplo de recuperação das configurações de envio de *e-mail* cujo parâmetro retornado é um mapa com outros valores necessários para configuração.

```
public MailThread(FacadeDelegate facade) {

    this.facade = facade;

    Map<String, String> paramMail = ParametroHelper.getInstance().
            getParametroMap(ConstantesParametroGeral.CONFIGURACOES_ENVIO_EMAIL);

    host = paramMail.get("host");
    username = paramMail.get("usuario");
    password = paramMail.get("senha");
    porta = paramMail.get("porta");
    from = paramMail.get("enderecoRemetente");
    fromName = paramMail.get("nomeRemetente");
    reply = paramMail.get("enderecoResposta");
    ssl = Boolean.valueOf(paramMail.get("usarSsl"));
    autenticacao = Boolean.valueOf(paramMail.get("autenticacao"));

}
```

Figura 3-12 Parâmetro com valores múltiplos

3.4.3 Uso Integrado das Técnicas

A combinação das técnicas composicionais e anotativas pode ser usada para implementar variabilidades. Através do uso dos padrões é possível implementar as variabilidades de granularidade grossa, habilitando que macro comportamentos sejam abstraídos na LPS. Já a técnica de execução condicional é usada para habilitar características opcionais ou para comportamentos distribuídos no código, as variabilidades de granularidade fina.

Na LPS SIGAA, um exemplo de *feature* alternativa opcional é a punição de usuários por atraso em empréstimos na entrega de livros ou outros materiais informacionais. Caso o usuário extrapole o prazo previsto ele deverá receber punições, que podem ser de dois tipos: (i) pagamento de multa em dinheiro; (ii) suspensão por um tempo para contrair novos empréstimos.

A LPS SIGAA permite habilitar estes comportamentos, usando as técnicas composicionais, como o padrão *Strategy*, para abstrair o comportamento geral das funcionalidades de punições e as técnicas anotativas para habilitar as *características* e escolher o tipo de estratégia concreta a ser usada. A Figura 3-13 ilustra este exemplo:

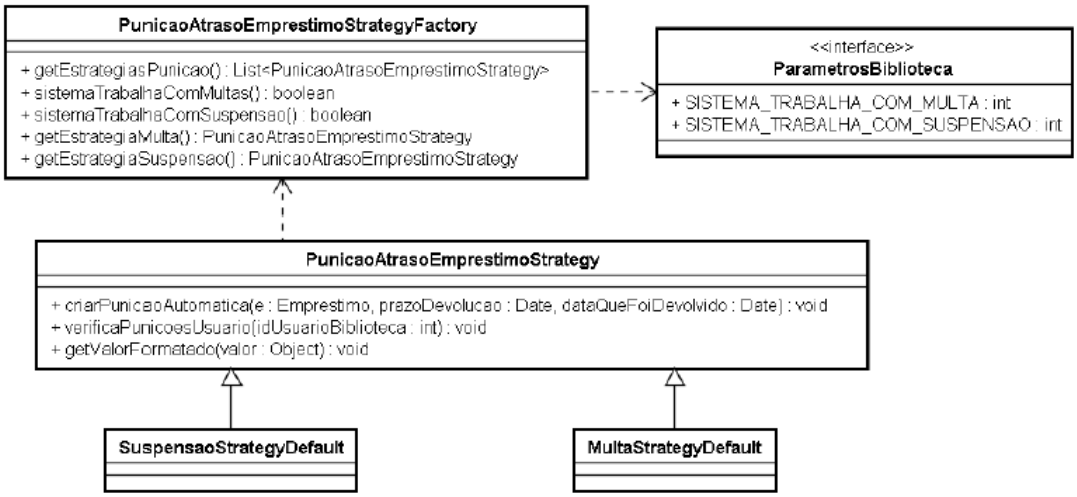


Figura 3-13 Variabilidade de Punição no módulo de bibliotecas do SIGAA

Os parâmetros *SISTEMA_TRABALHA_COM_MULTA* e *SISTEMA_TRABALHA_COM_SUSPENSAO*,*NOME_CLASSE_IMPLEMENTA_ESTRATEGIA_SUSPENSAO* e *NOME_CLASSE_IMPLEMENTA_ESTRATEGIA_MULTA* são usados nas técnicas anotativas.

As implementações de *características* alternativas e opcionais podem ser usadas com os padrões *Strategy, Factory Method* e *Template Method*, todas em conjunto com a parametrização para escolha da variante. A implementação de características *OR-Feature* pode ser usada com o padrão *Chain Of Responsability*.

3.5 Impacto das Técnicas na Evolução do Software e Conflitos

Uma das principais motivações para modularização das variabilidades dos sistemas SIG é permitir uma evolução robusta da LPS evitando ao máximo a criação de comportamentos conflitantes em produtos que inviabilizem a reconciliação dos artefatos de código do núcleo e dos produtos da LPS. As técnicas expostas são aplicadas tanto para os cenários de alta quanto baixa granularidade de mudanças.

As técnicas anotativas devem ser criadas, prioritariamente, nos artefatos da linha de produto a ser clonada, denominada *source*, pois, a adaptação de métodos para incorporação de novos parâmetros de granularidade fina no clone (*target*), em geral, gerará conflitos entre o *source* e o *target*. Esta técnica deve ser aplicada em variabilidades de características opcionais. Ela também pode ser utilizada para

características alternativas, mas, neste caso é preciso ter certeza que as variantes definidas tem poucas chances de mudanças no *target*.

As técnicas composicionais são mais robustas e menos propensas a conflitos. Elas devem ter seu uso incentivado, uma vez que permitem que as evoluções em variabilidades do *target* não conflitem com as evoluções do *source*, possivelmente facilitando a reconciliação das LPS. As técnicas composicionais também podem ser aplicadas para variações de granularidade fina, desde sejam usados padrões de projetos adequados. O uso do *Template Method*, por exemplo, pode ser aplicado para inclusão de métodos abstratos na parte do fluxo de código que possui comportamento variável e, então, implementá-los através de subclasses. O capítulo 6 apresenta um estudo empírico que analisa possíveis conflitos de integração de código encontrados quando reconciliando clones das LPS SIGAA e SIPAC clonadas que utilizam técnicas composicionais e anotativas para implementar suas variabilidades.

3.6 Sumário

Ao longo deste capítulo foram apresentados conceitos relacionados às linhas de produto de software para sistemas de informações web, especificamente como caso de aplicação dos sistemas de informações SIGAA e SIPAC. Estes sistemas possuem um conjunto de variabilidades que foram catalogadas nas Tabela 3-2 e Tabela 3-3. A arquitetura multicamadas de desenvolvimento de aplicações foi apresentada, assim como as variabilidades se relacionam com as camadas de software. Em seguida, foram abordadas as técnicas para implementação de variabilidades, dividindo-as em composicionais e anotativas. As técnicas composicionais são usadas principalmente com padrões de projetos, tendo sido exemplificado o uso dos principais padrões relacionados: *Factory Method, Strategy, Chain of Responsability e Template Method*. As técnicas anotativas abordam a execução condicional como forma recomendada para implementação de variabilidades de granularidade fina. Por fim, foi analisado o impacto do uso dessas técnicas, relacionando-as com a evolução de LPS.

4 Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos

Este capítulo apresenta uma abordagem para evolução e reconciliação de linhas de produtos de software clonadas. A seção 4.1 aborda o problema de evolução de evolução de LPS clonadas. A seção 4.2 apresenta uma visão geral da abordagem, descrevendo seus principais elementos. A seção 4.3 detalha os passos da abordagem.

4.1 Evolução e Reconciliação de Produtos Clonados

A utilização da técnica de clonagem de produtos para fins de reutilização é motivada principalmente pela flexibilidade de mudanças que cada produto clonado permite, bem como a economia de tempo e recursos para iniciar um projeto a partir de um conjunto de artefatos já verificados (Dubinsky et al., 2013). Cada usuário ou empresa interessada em criar um novo produto (sistema) da linha de produto de software (família de sistemas) modifica este novo produto em razão de seus interesses e com equipe de desenvolvimento própria, criando novas versões de código fonte independentes e adaptadas à realidade da organização. Ao mesmo tempo em que o clone derivado evolui, a LPS que originou o clone também evolui gerando novas características. Tanto as evoluções feitas na origem do clone como as da LPS clonada podem ser de interesse mútuo.

A Figura 4-1 ilustra este cenário, observe que após a derivação inicial cada uma das versões evolui independentemente (representado por cada coluna na figura), tornando as versões incompatíveis entre si. Neste cenário, caso os clones 1 e 2 queiram recuperar características publicadas na versão 4.1 da LPS *source*, eles podem apenas fazer uso das técnicas existentes de *merge* de controle de versão em repositórios, tornando o processo lento, custoso e sujeito a erros, como a maioria da análise de conflitos é feita manualmente pelo desenvolvedor.

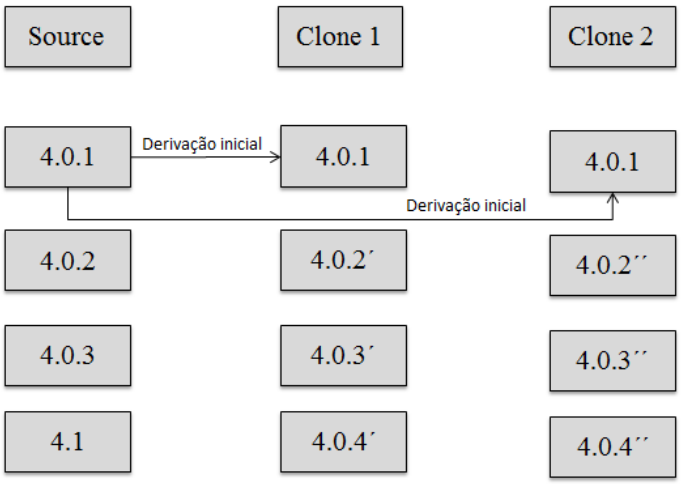


Figura 4-1 Evolução concomitante de produtos

4.2 A Abordagem de Reconciliação de Produtos Clonados

A abordagem proposta nesta tese tem como objetivo: (i) permitir que os clones evoluam de forma independente; e a partir da necessidade de reconciliação, (ii) detectar e indicar os tipos de conflitos existentes entre versões independentes derivadas de uma versão de código-fonte de uma LPS; (iii) prover mecanismos para realizar o *merge* – automatizado ou semi-automatizado – dos conflitos existentes; e (iv) promover análises e reconciliações orientadas a tarefas. A Figura 4-2 ilustra o funcionamento geral da abordagem, onde *source* representa a LPS original, e o *target* representa a LPS clonada em uma nova versão independente de código. O objetivo da abordagem proposta é permitir que novas características desenvolvidas na LPS *source* sejam incorporadas na LPS *target*.

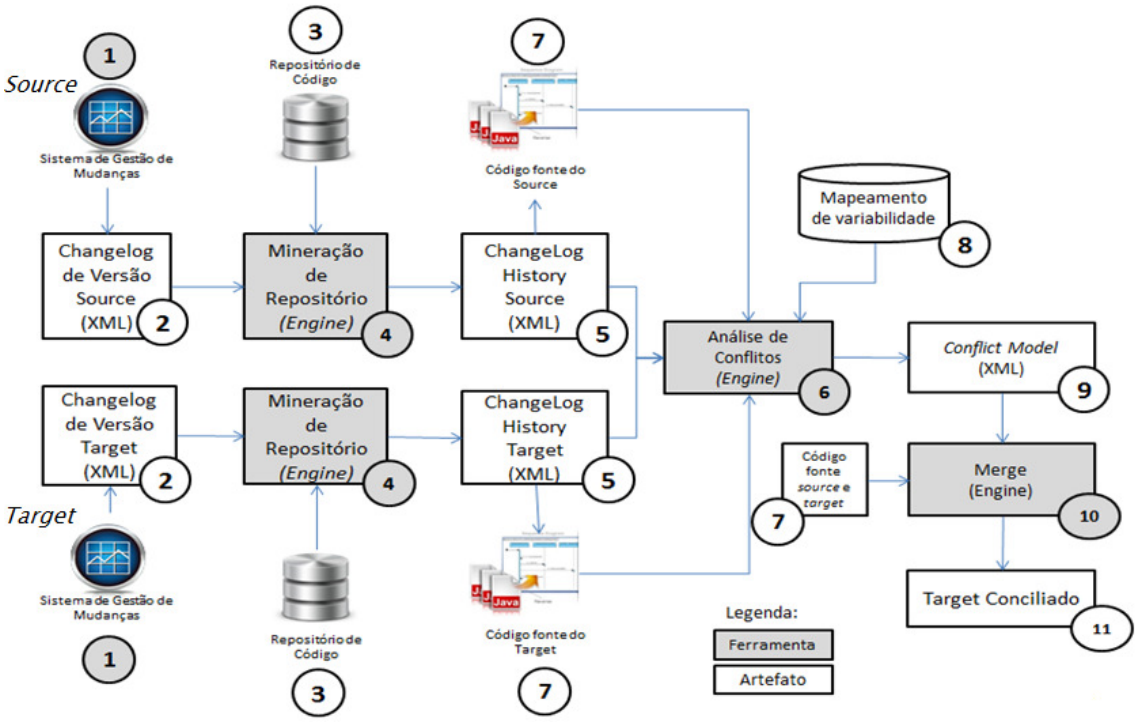


Figura 4-2 Estrutura geral da abordagem e seus elementos

Para facilitar a compreensão da abordagem, antes de detalhar seu fluxo de execução é necessário listar cada um dos seus componentes e sua respectiva função:

- **1 - Sistema de gestão de mudanças (*issue tracking system*):** estes componentes são responsáveis por gerenciar as respectivas tarefas de mudanças realizadas no *software*, associando as mesmas às respectivas versões/revisões de arquivos de artefatos de implementação (código-fonte), tanto nas evoluções realizadas no *source* quanto no *target*. Este sistema representa um componente fundamental na abordagem, pois é a partir dele que os arquivos de versões com respectivas mudanças e as revisões do repositório são obtidas. Exemplos de tais sistemas são: *Bugzilla*, *ClearQuest* e *Redmine*. No caso especifico dos projetos da UFRN e SIG Software são usados os sistemas iProject e SIGProject, respectivamente.

- **2 - Arquivo de *changelog* de versão:** arquivo XML contendo as modificações realizadas na versão da LPS em análise. Este arquivo possui as seguintes informações: nome do sistema, versão inicial e final das mudanças que estão no arquivo e lista das tarefas modificadas. Cada uma das tarefas possui as seguintes informações: número, descrição resumida, descrição detalhada, tipo da evolução,

módulo e lista de publicações (revisões) no controle de versões com seus respectivos artefatos.

- **3 - Repositório de código:** o repositório representa o sistema de controle de versões das LPS sob análise. A versão atual da abordagem oferece suporte ao SVN, porém a abordagem é genérica e pode funcionar com qualquer tipo de repositório, seja ele centralizado ou distribuído.

- **4 - Mineração de Repositório:** a mineração de repositório é um componente da abordagem que através das revisões (*commits*) contidas no arquivo de *changelog* de versão, se conecta ao repositório para descobrir, em detalhes, que mudanças os artefatos tiveram. A implementação atual da abordagem permite lidar com as seguintes mudanças: (i) adição, remoção e atualização de classes; (ii) adição, remoção e atualização de métodos; (iii) adição, remoção e atualização de atributos; e (iv) atualização de estruturas de classes, em termos de novas relações de heranças e/ou interfaces.

- **5 - *Changelog history:*** o *changelog history* é um modelo gerado pelo componente de mineração de repositório, gerado em memória e exportado para um arquivo XML. Ele representa um arquivo de *changelog* de versão, enriquecido com as informações obtidas pela mineração, e que estão relacionadas às evoluções dos elementos de implementação.

- **6 - Analisador de Conflitos:** o analisador de conflitos é o componente da abordagem que efetua a comparação entre o *changelog history* do *source* e do *target*, e identifica os conflitos diretos e textuais de código que ocorreram como resultado do *merge* das evoluções dos projetos. A identificação de conflitos indiretos é realizada através de análise do grafo de chamadas do código-fonte. Cada um destes conflitos será detalhado nas seções seguintes deste capítulo.

- **7 - Código Fonte:** os códigos fonte das LPS são usados para a análise de conflitos indiretos e para verificar se a evolução ocorreu no núcleo ou nas variabilidades da LPS. O código fonte do *target* também é usado pelo *merge*

engine para a reconciliação do código fonte na aplicação das tarefas selecionadas do *source*.

- **8 - Arquivo de mapeamento de variabilidades:** arquivo que armazena o mapeamento de variabilidades para artefatos de código fonte da LPS que está sendo analisada. Essa informação é usada pela abordagem para indicar se o conflito ocorreu no núcleo da LPS ou em alguma de suas variabilidades.

- **9 - *Conflict Model*:** após a análise dos conflitos a abordagem gera um modelo contendo os conflitos, os seus tipos e em qual evolução eles ocorrem no *source* e no *target*.

- **10 - *Merge Engine*:** o motor de execução do *merge* é o componente que percorre as tarefas selecionadas no *source* e realiza a análise e reconciliação para sua aplicação no *target*, auxiliando na resolução dos conflitos, através da realização de *merges* automatizados ou através da indicação de possíveis conflitos que auxiliem os engenheiros de *software*, durante a realização de um *merge* semi-automatizado ou manual.

- **11 - *Target* Conciliado:** representa o objetivo final da abordagem, ou seja, o código fonte do *target* conciliado contendo o conjunto de evoluções do *source* que foram selecionadas pelo engenheiro para serem integradas ao *target*.

A abordagem pode ser sintetizada em quatro macro-passos conforme ilustra a Tabela 4-1:

	Operação	**Entrada**	**Saída**
1	Geração do arquivo de evoluções de versão (*changelog*)	Sistema de gestão de mudanças	Arquivo de *Changelog* de Versão (XML)
2	Mineração do repositório de código	Arquivo de *Changelog* Repositório de código	Arquivos de *Changelog* History

3	Análise de conflitos	Arquivo de *Changelog history*, Arquivo de mapeamento de variabilidades, código fonte	*Conflict Model*
4	*Merge*	*Arquivos de History Changelog*, *Conflict Model* e Código fonte	Código fonte do *Target* Conciliado

Tabela 4-1 Passos para execução da abordagem de reconciliação

4.3 Geração dos Arquivos de Evoluções

A geração do arquivo de evoluções de *software* é o primeiro passo da abordagem, pois é o ponto de partida para obtenção de cada mudança realizada na LPS (ou algum de seus produtos clonados) de forma agrupada por tarefa. Neste passo, utilizando uma ferramenta de gestão de mudanças, são registradas todas as tarefas de manutenção com seus respectivos tipos e artefatos relacionados, e então, a partir do conceito de uma versão disponível ao usuário (*release*), gera-se um relatório, em formato XML, contendo um descritivo analítico com as informações de cada tarefa.

A Tabela 4-2 ilustra as informações da estrutura do arquivo. A Figura 4-3 ilustra um exemplo de arquivo de *changelog*, em formato XML, contendo as informações de uma determinada versão do sistema SIPAC. Pode-se observar que este arquivo contém informações sobre gestão de mudanças, que podem ser obtidas através de ferramentas tradicionais de desenvolvimento de sistemas.

Cabeçalho do arquivo

Informação	**Descrição**
Sistema	Nome do sistema cujo arquivo contém as evoluções
Versão Inicial	Versão inicial a partir da qual as mudanças foram selecionadas
Informação contida em cada tarefa	
Versão Final	Versão final até onde as mudanças foram selecionadas
Informação	Descrição
Número	Número de controle da tarefa utilizada pela ferramenta de desenvolvimento de *software*.
Nome da tarefa	Informação resumida indicando a que se refere a tarefa.
Descrição da tarefa	Informações detalhadas da tarefa
Tipo de Tarefa	Categoria da evolução: Nova Operação, Evolução de existentes ou correção de *bugs*.
Status	Descrição do status da tarefa (Finalizada, Suspensa, dentre outras)
Percentual	Percentual de cumprimento da tarefa
Versão	Versão em que a tarefa foi publicada. Neste caso, esta informação é relevante uma vez que um arquivo de *changelog* pode conter evoluções de diversas versões agregadas.
Módulo	Módulo ou subsistema do sistema a qual a tarefa se refere.
Publicações no repositório	Lista de publicações no repositório relacionadas com esta tarefa. É a partir deste campo que se vincula os artefatos com a tarefa.

Tabela 4-2 Estrutura de dados do arquivo de *changelog*

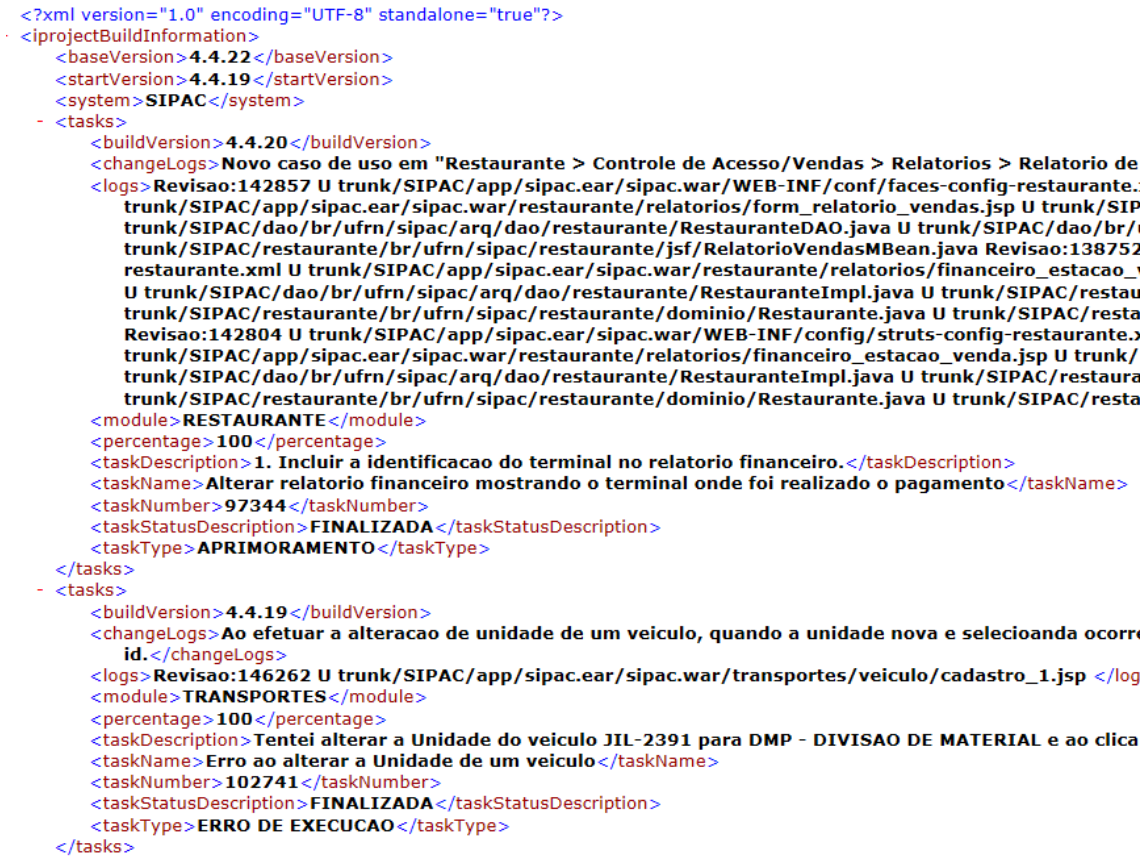


Figura 4-3 Fragmento de arquivo de *changelog* de versão em formato XML

4.4 Mineração de Repositório de Código

A mineração de repositório tem como objetivo analisar informações disponíveis em repositórios de *software* (sistemas de controle de versões, sistema de gerência de mudanças, repositórios de e-mails, dentre outros), com o objetivo de descobrir informações relevantes sobre o processo de engenharia de software (Herzig & Zeller, 2011).

Observe que no arquivo de *changelog* de versão, ilustrado na Figura 4-3, o elemento XML `logs` contém os arquivos modificados por cada tarefa. Porém, por se tratar apenas de uma listagem de revisões de código do repositório, tal lista não indica especificamente o que foi modificado. Para realizar a análise de conflitos é necessário obter detalhes de cada evolução das classes do sistema através da mineração de repositório de código, de forma a identificar o estado anterior e o atual para compará-los e extrair atomicamente as seguintes operações:

- CLASS ADD: adição de classe;
- CLASS REMOVE: remoção da classe;
- CLASS UPDATE: mudanças específicas dos elementos das classes sem considerar seus atributos e métodos. Neste caso, consideram-se as declarações de herança (`extends`) e interfaces (`implements`);
- FIELD ADD, REMOVE: adição e remoção de atributos;
- FIELD UPDATE: atualização de atributos em relação aos seus valores de inicialização. Mudanças de tipos do atributo são tratadas como remoção do atributo anterior e adição de um novo;
- METHOD ADD, METHOD REMOVE: adição e remoção de métodos;
- METHOD UPDATE: atualização do corpo do método com mudanças no código ou declarações de exceções;
- As operações de mudança de nome (*rename*) são tratadas como DELETE e ADD.

Após a execução da mineração, o resultado é a geração de um histórico de versão enriquecido com informações adicionais, chamado de *History Changelog*. Trata-se de um arquivo XML, contendo as informações relevantes do *changelog* de versão com os dados das evoluções mineradas. A Figura 4-4 ilustra um arquivo de *history changelog*. Neste arquivo, estão contidas as informações obtidas da mineração do repositório de códigos-fonte e do sistema de gestão de demandas agrupados por cada tarefa (*changelog*). Observe que apesar de não utilizado nesta tese o modelo suporta o relacionamento entre as tarefas (*changelog*) e as *features* do modelo de característica através da *tag* `feature`.

```
<changeloghistory>
  <system>SIPAC</system>
  <startVersion>4.4.19</startVersion>
  <baseVersion>4.4.22</baseVersion>
  <features>

    <feature>
      <name>CORE</name>
      <type>MANDATORY</type>
      <description>THE CORE FEATURE</description>
      <changelogs>
        <changelog>
          <identify>97344</identify>
          <type>UPGRADING</type>
```

```
          <version>4.4.20</version>
          <description>Alterar relatorio financeiro mostrando o terminal onde foi
realizado o pagamento</description>
          <changeInformation>Novo caso de uso em &quot;Restaurante &gt; Controle de
Acesso/Vendas &gt; Relatorios &gt; Relatorio de Vendas&quot;.</changeInformation>
          <module>RESTAURANTE</module>
          <classes>
            <class>
<signature>dao/br/ufrn/sipac/arq/dao/restaurante/
RestauranteDAO.java</signature>
              <directlyConflicting>false</directlyConflicting>
              <indirectlyConflicting>false</indirectlyConflicting>
              <textualConflicting>false</textualConflicting>
              <changesLocation>
                   <ChangeLocation>BODY</ChangeLocation>
              </changesLocation>
              <name>RestauranteDAO.java</name>
<path>trunk/SIPAC/dao/br/ufrn/sipac/arq/dao/restaurante/RestauranteDAO.java</path>
              <changeType>U</changeType>
              <revision>142857</revision>
              <fields/>
              <methods>
                <method>
<signature>Collection&lt;PagamentoRefeicao&gt;#findRelatorioRefeicoesPagas(Date,Date,Int
eger,Integer)</signature>
                  <directlyConflicting>false</directlyConflicting>
                  <indirectlyConflicting>false</indirectlyConflicting>
                  <textualConflicting>false</textualConflicting>
                  <changesLocation>
                    <ChangeLocation>PARAMETERS</ChangeLocation>
                  </changesLocation>
                  <name>findRelatorioRefeicoesPagas</name>
                  <changeType>U</changeType>
                </method>

              </methods>
              <annotations/>
              <implementations/>
            </class>
            <class>
              <signature>restaurante/br/ufrn/sipac/restaurante/dominio/
              EstacaoVenda.java</signature>
              <directlyConflicting>false</directlyConflicting>
              <indirectlyConflicting>false</indirectlyConflicting>
              <textualConflicting>false</textualConflicting>
              <changesLocation>
                   <ChangeLocation>BODY</ ChangeLocation>
              </changesLocation>
              <name>EstacaoVenda.java</name>
<path>trunk/SIPAC/restaurante/br/ufrn/sipac/restaurante/dominio/EstacaoVenda.java</path>
              <changeType>U</changeType>
              <revision>138752</revision>
```

```
<fields>
  <field>
    <signature>int:quantidade</signature>
    <directlyConflicting>false</directlyConflicting>
    <indirectlyConflicting>false</indirectlyConflicting>
    <textualConflicting>false</textualConflicting>
    <name>quantidade</name>
    <changeType>A</changeType>
    <classChangeLog reference="../../.."/>
    <annotations>
      <annotation>
        <signature>Transient</signature>
        <directlyConflicting>false</directlyConflicting>
        <indirectlyConflicting>false</indirectlyConflicting>
        <textualConflicting>false</textualConflicting>
        <name>Transient</name>
        <changeType>A</changeType>
      </annotation>
    </annotations>
  </field>

 ...
```

Figura 4-4 Arquivo de *History Changelog*

4.5 Análise e Caracterização de Conflitos

Os conflitos entre evoluções ocorrem quando uma determinada evolução (*changelog*) no *source* conflita com outra evolução no *target*, dificultando ou impedindo a reconciliação de versões. De acordo com Yuriy (Yuriy, 2013), os conflitos são custosos porque trazem um atraso ao projeto, para que o engenheiro de *software*, responsável por integrar as mudanças realizadas, possa entendê-los e então solucioná-los. O receio de gerar conflitos no código também traz custos, pois pode levar os desenvolvedores a postergar sua modificação, aumentando o potencial de modificações concorrentes sob os mesmos artefatos.

LPS *Source*	LPS *Target*

`public class Calc {`
 	
`int result = 0;`
	`public class Calc {`
	
	`int result = 0;`

```
        public void add(int a,int b) {

                result  = a + b;

        }

        public void div(int a, int b) {

                result = a / b;

        }

}
```

```
        public void add(int a,int b) {

                result  = a + b;

        }

      public void mult(int a, int b) {

                result = a * b;

        }

}
```

Figura 4-5 Exemplo de conflito textual

Nossa abordagem caracteriza os conflitos como sendo: (i) textuais, (ii) diretos e (iii) indiretos. Para exemplificar um conflito, considere o seguinte cenário de evolução entre o *source* e *target* ilustrado na Figura 4-5. É possível observar que a evolução da LPS *source* conduziu à criação de uma nova operação na classe `Calc` (método `div`). Por sua vez, a evolução da LPS *target* conduziu à criação de uma nova operação de multiplicação (o método `mult`). Nas técnicas tradicionais de desenvolvimento de *software*, com uso de controles de versões, este conflito deveria ser tratado manualmente pelo desenvolvedor, pois evoluiu na mesma fração de texto nos dois artefatos. Este conflito representa um conflito léxico, porém não semântico, pois as classes evoluíram em métodos distintos e sem correlação entre si. A abordagem proposta nesta tese caracteriza este tipo de conflito como **conflito textual**. Os conflitos textuais representam conflitos leves, pois apesar dos artefatos evoluírem concomitantemente, a evolução ocorreu em métodos e/ou atributos distintos, o que pode caracterizar evoluções passíveis de resolução automatizada de conflitos.

LPS *Source*	LPS *Target*

`public class Calc {`
`  int result = 0;`
`  public void add(int a,int b) {`
`  result  = a + b;`
`  Log.gravaLog(result);`
`  }`
	`public class Calc {`
	`  int result = 0;`
	`  public void add(int a,int b) {`
	`  result  = a + b;`
	`  if ( result > 0 )`
	`  Log.writeXML(result);`

} 	} }

Figura 4-6 Exemplo de conflito direto

A Figura 4-6 ilustra evoluções do *source* e do *target* que modificam o mesmo método da classe `Calc`, cada um com o seu propósito. Neste cenário, os artefatos conflitaram diretamente, tanto léxica como semanticamente. A este tipo de conflito, onde há modificações distintas realizadas no mesmo método ou atributo, tanto no *source* como no *target*, denomina-se **conflito direto**.

Os conflitos diretos e textuais representam mudanças concomitantes na mesma classe. Por ser uma mudança explícita ocorrida nos mesmos artefatos do *source* e *target*, são visíveis ao desenvolvedor. No entanto, muitos dos erros inseridos em modificações de *software* decorrem dos impactos oriundos de modificações em classes e métodos dependentes, ou seja, mudanças que ocorrem em uma determinada funcionalidade, mas que por ser usada por outra, acaba tendo impacto ao provocar um funcionamento não desejado. A seguir ilustram-se alguns exemplos dos conflitos chamados indiretos. A Figura 4-7 ilustra o método `matricular()` da classe `MatriculaBean` o qual possui uma chamada para o método `findDiscentes()` da classe `MatriculaDao` que retorna os discentes que estão pendentes de matrícula para uso na lógica de controle/negócio da aplicação. A Figura 4-8 apresenta o fluxo de dependência do método `matricular()`, ilustrando sua dependência do retorno esperado do método `findDiscentes()`.

```
public class MatriculaMBean {

        public String matricular() {
                MatriculaDao dao = DAOFactory.getDAO(Matricula.class);
                List<Discente> pendentesMatricula = dao.findDiscentes();
                ...
        }
}
```

Figura 4-7 Exemplo de Conflito Indireto (i)

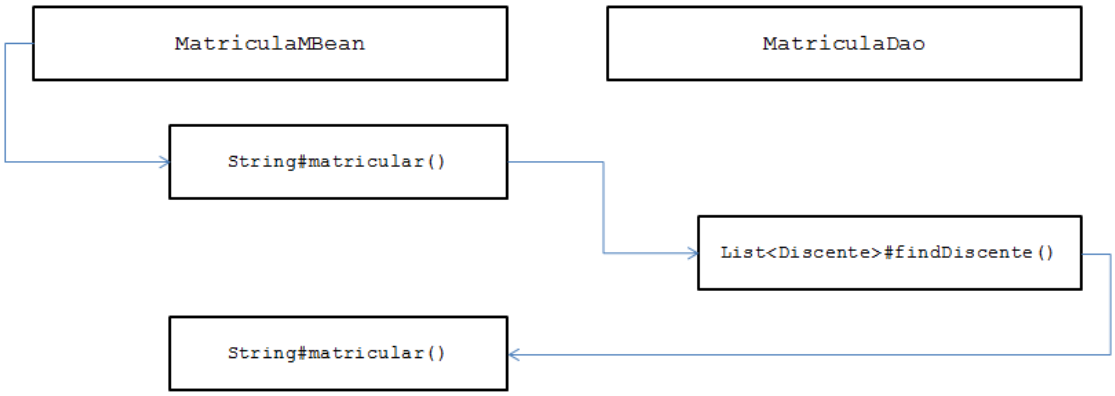


Figura 4-8 Fluxo de chamada do exemplo de conflito indireto

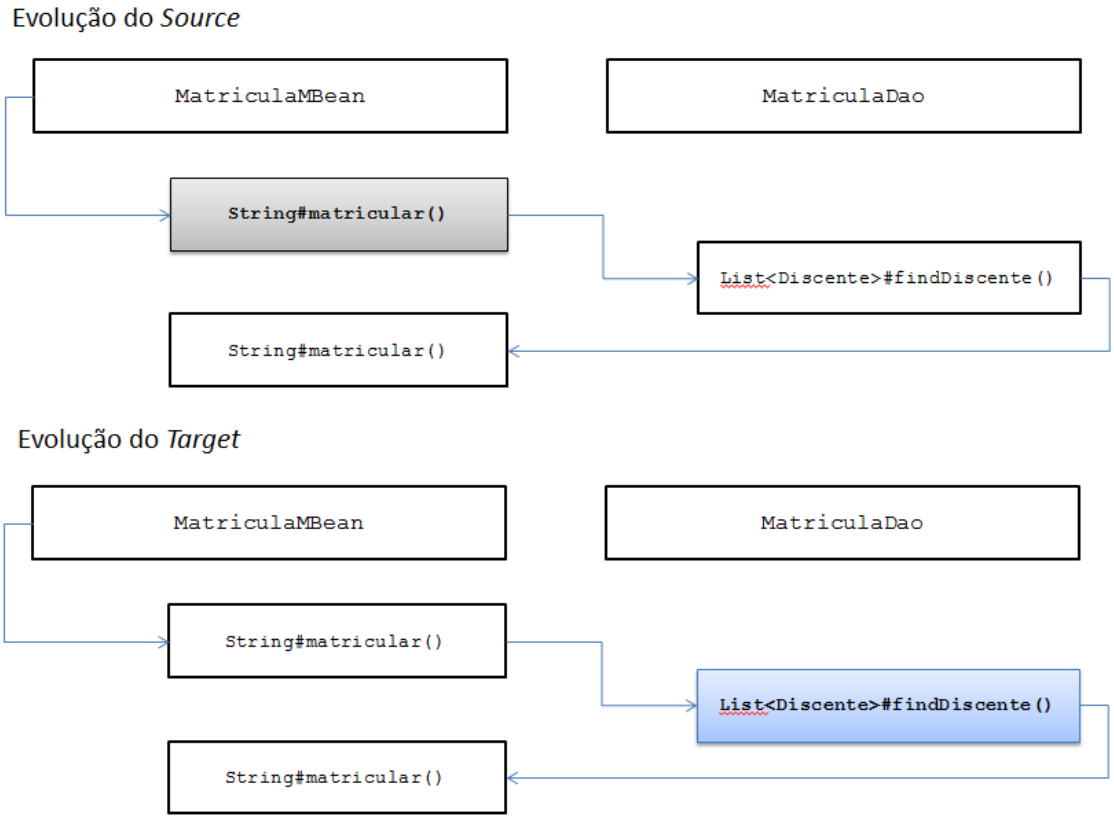


Figura 4-9 Evolução geradora de conflito indireto

A Figura 4-9 apresenta um exemplo de conflito indireto no contexto de evolução do método `matricular()` da classe `MatriculaBean`. A LPS *source* sofreu uma evolução no método `matricular()` da classe `MatriculaMBean`, e em paralelo, a LPS *target* evoluiu o método dependente `findDiscentes()`. Neste caso, trata-se de um

conflito indireto, pois o *target* evoluiu um artefato (método `findDiscentes`) que faz parte do grafo de chamadas do artefato evoluído do *source* (método matricular). Esta indicação não representa necessariamente um problema, pois a evolução da busca de discente no *target* pode ter sido realizada para atender a um cenário de negócio necessário do usuário do *target*, porém a abordagem deve alertar o engenheiro de *software* sobre este conflito, para que o mesmo possa decidir sobre a incorporação automatizada da evolução. Este conflito na nossa abordagem é denominado **conflito indireto dependências**, ou seja, a aplicação de uma tarefa do *source* teve algum método dependente do grafo de chamadas modificado no *target*.

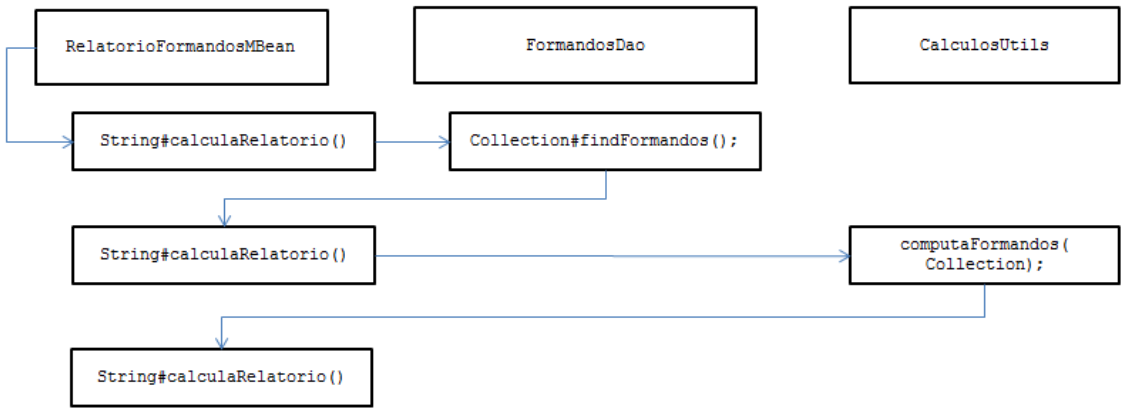


Figura 4-10 Evolução e Grafo de Chamada - Conflito Indireto Dependências

A Figura 4-10 ilustra o grafo de chamadas de uma nova operação criada no *target:* a emissão do relatório de formandos. Para emissão do relatório o método `calculaRelatorio()` chama o método `Collection#findFormandos()`, e logo em seguida, chama o método `computaFormandos()` da classe `CalculosUtils`. Conforme se pode observar, o método `computaFormandos()` é chamado ao longo do grafo de chamadas de uma operação do *target*. Considere-se agora que na reconciliação de uma versão do *source* um dos *changelogs* possui a seguinte evolução atômica:

```
Method - UPDATE - Collection#computaFormandos() - CalculosUtils;
```

A incorporação desta evolução na versão do *target* pode representar um mau funcionamento do novo relatório criado, pois não se sabe se o método `computaFormandos()` foi modificado de forma a afetar as mudanças realizadas pelo *target*. Desta forma, se caracteriza um conflito indireto de uma operação do grafo de chamadas do *target* que foi modificada pelo *source*. Este tipo de conflito é denominado de **conflito indireto referências,** ou seja, a aplicação de uma tarefa do *source* contém modificação de algum método que o grafo de chamadas do *target* depende e foi modificada pelo *source*.

Diferentemente dos conflitos diretos e textuais, que possuem o seu algoritmo de detecção restrito às análises das evoluções do *source* e *target* obtidas da mineração do repositório e contidos nos arquivos de *history changelog*, a busca dos conflitos indiretos necessita realizar cruzamentos entre o código fonte da aplicação e as evoluções atômicas. Na análise dos conflitos indiretos dependentes avalia-se no grafo de chamada (*call graph*) das modificações do *source* se nenhum deles foi modificado por alguma tarefa do *target*. Já a análise de conflitos indiretos referências avalia se no grafo de chamadas de uma operação do *target* se a reconcilia não modificará nenhum método contido no grafo de chamadas. As Figura 4-11 e Figura 4-12 apresentam, respectivamente, os algoritmos de detecção de conflitos indiretos dependentes e conflitos indiretos referências.

```
Para cada evolução atômica no source
   Localizar o método/atributo no código fonte
   Pesquisar, resursivamente, os artefatos (métodos e atributos)
   dependentes (call graph) considerando profundidade
   configurada.
   Para cada artefato dependente
         Verificar se o artefato teve evolução no target.
               Caso sim
                  Marcar como Conflito Indireto
                  Adicionar conflito no conflict Model
```

Figura 4-11 Algoritmo de Conflitos Indiretos Dependentes

```
Para cada evolução atômica no source
   Localizar o método/atributo no código fonte
   Pesquisar, recursivamente, os artefatos (métodos e atributos)
   referenciados (references) considerando profundidade
   configurada.
   Para cada artefato dependente.
         Verificar se o artefato teve evolução no target.
               Caso sim
                  Marcar como Conflito Indireto
                  Adicionar conflito no conflict Model
```

Figura 4-12 Algoritmo de Conflitos Indiretos Referências

Os conflitos indiretos representam uma contribuição importante para a abordagem pois são difíceis de serem detectados pelo engenheiro de *software*, considerando que exigem a análise de dezenas a centenas de artefatos que foram modificados no código do *source* e do *target*. Estes conflitos nem sempre representam problemas, mas auxiliam para que o *merge* possa ser realizado de forma mais segura e trazendo o mínimo de impacto possível na reconciliação. Em via de regra, o alerta emitido pelos conflitos indiretos indica para nossa abordagem que as funcionalidades modificadas no *source* e no *target* são dependentes e, portanto, precisam ter seus testes executados novamente.

Há outros tipos de conflitos indiretos mais complexos de serem detectados pela análise estática, como por exemplo: a mudança de valor de um atributo tempo de execução que interfere na execução de um determinado método. Alterações em arquivos ou dados compartilhados, dentre outros. Estes conflitos indiretos não podem ser detectados pela análise estática e exigira uma estratégia mais robusta de análise dinâmica em tempo de execução para detectá-los.

4.5.1 Resumo dos Tipos de Conflitos

Um conflito de código é identificado quando um par de mudanças realizadas no *source* e no *target*, respectivamente, possuem algum tipo de incompatibilidade léxica,

estrutural ou semântica, quando estiverem sendo integradas em um processo de reconciliação de uma LPS ou de produtos de uma LPS.

Os conflitos podem ser categorizados nos seguintes tipos:

1) **Textual (Léxico)**: uma mudança no *source* e uma no *target* é realizada sobre a mesma classe, cada um em sua cópia, mas sobre diferentes elementos estruturais que não dependem um do outro.

2) **Direto (Estrutural)**: uma mudança de código do *source* e uma mudança de código do *target* alteram (criam, modificar, removem) o mesmo elemento de implementação (atributo ou método).

3) **Indireto (Semântico):** uma mudança de código do *source* conflita com uma mudança de código do *target*, de forma que a mudança do *source* ocorre no grafo de chamadas do elemento do *target* ou uma mudança no *target* ocorre no grafo de chamadas do elemento do *source*. O primeiro é denominado de Conflito Indireto Referências e o segundo conflito indireto Dependências.

4.6 Merge Engine

Após a execução da análise dos conflitos a abordagem identifica quais tarefas podem ser conciliadas: (i) sem nenhum tipo de conflito; (ii) aquelas que possuem conflitos que podem ser resolvidos automaticamente; (iii) as que podem ser resolvidas de forma semi-automatizada; e (iv) as que exigem a intervenção do engenheiro para sua resolução. O *merge* é a principal etapa para fins de reconciliação da LPS clonada, pois é ele que irá injetar no código fonte do *target* as mudanças ocorridas no *source*.

A reconciliação pode ocorrer em quatro cenários:

1) **Ausência de Conflitos:** as tarefas foram identificadas pelo analisador de conflitos como seguras para serem conciliadas e o *merge* realizará a incorporação no *target* dos artefatos relacionados do *source*.

2) **Resolução Automatizada de Conflitos:** situação onde apesar de ocorrer o conflito, a ferramenta procede com sua resolução de forma automatizada. Este cenário ocorre com os **conflitos textuais** detectados pela abordagem.

3) **Resolução Semi-automatizada de Conflitos:** situação onde o *merge* pode incorporar as tarefas sem intervenção do engenheiro, mas a abordagem indica as tarefas ou trechos de códigos que precisam ser re-testadas para garantir que a integração de código modificado pelo *source* e pelo *target* continue funcionando. Este cenário ocorre nos **conflitos indiretos**.

4) **Resolução manual de conflitos:** situação na qual não há possibilidade de *merge* automatizado, nem como determinar de forma segura as implicações que a alteração terá no produto. Assim, é necessária a intervenção manual do engenheiro para sua correção. Este cenário ocorre nos **conflitos diretos**.

A execução do *Merge Engine* segue o algoritmo ilustrado na Figura 4-13. O resultado final do processo de *merge* é o *target* conciliado com as tarefas selecionadas do *source*, permitindo assim que o clone possa receber mudanças do *source* de forma segura e continuar sua evolução.

```
Para cada tarefa selecionada no source para reconciliação
        Seleciona cada classe e constrói a árvore sintática abstrata (AST)
        Constrói a AST de cada artefato correspondente do Target
        Identifica as operações atômicas do source que devem ser injetadas
        Inicia uma transação de mudança de artefatos no target
                Injeta cada operação atômica do source na AST do target
                Atualiza a classe com a AST modificada
                Indica operações a serem testadas, caso existam.
```

```
Testa se compilação ocorreu com sucesso.

        Caso sim:  Finaliza transação escrevendo mudanças;

        Caso não:  reverte modificação avisando os erros
```

Figura 4-13 Algoritmo de *Merge*

4.7 Sumário

Ao longo deste capítulo foram apresentados os problemas relacionados à evolução concomitante de linhas de produto de software clonadas, contextualizando como tal evolução pode gerar conflitos durante operações de *merge* do código. A abordagem proposta nesta tese permite a reconciliação de LPS clonadas através de quatro passos: (i) extração das informações de versão; (ii) mineração do repositório; (iii) análise de conflitos; e (iv) *merge* do código. A execução de cada um desses passos foi detalhada para que se possa obter o resultado final, que é o *target* conciliado com as tarefas oriundas do *source*. O próximo capítulo detalha os aspectos relacionados ao projeto e à implementação descrita.

5 Projeto e Implementação da Abordagem

Este capítulo apresenta o projeto e implementação da abordagem de reconciliação de linhas de produtos. A Seção 5.1 apresenta a arquitetura e projeto geral de implementação da abordagem. A Seção 5.2 detalha a extração das informações de ferramentas de gestão de demandas e a mineração de repositório. A seção 5.3 detalha a análise de conflitos e a seção 5.4 o merge *engine*.

5.1 Arquitetura da implementação

A ferramenta de apoio à abordagem foi desenvolvida como um *plugin* do ambiente de desenvolvimento Eclipse, formado pelos componentes indicados a seguir. A Figura 5-1 apresenta a estrutura de tais elementos;

- ***Changelog Reader:*** classes auxiliares para leitura/gravação dos arquivos XML utilizados pela ferramenta: *changelog* e *history changelog*.
- ***Miner Engine***: componente responsável pela mineração de dados no repositório de classes.
- ***Conflict Analisys:*** componente responsável pela análise dos conflitos.
- ***Merge Engine:*** componente de execução do *merge* das tarefas selecionadas.
- ***Statistics Engine:*** componente responsável pela geração de estatísticas dos conflitos e da análise detalhada das evoluções em arquivos texto.
- ***Workspace:*** *workspace* com o código fonte do *source* e do *target* em projetos Java do Eclipse.

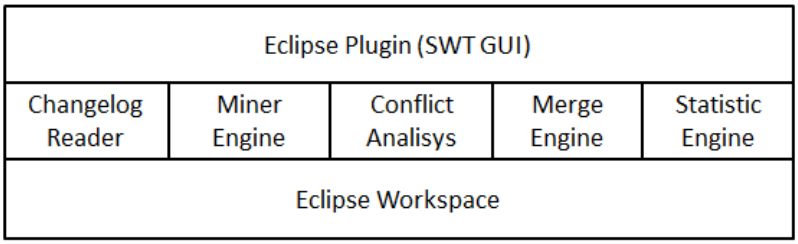


Figura 5-1 Arquitetura Geral da Ferramenta de Apoio a Abordagem

As configurações referentes aos endereços dos repositórios e nomes dos projetos a serem utilizados são realizadas no arquivo `config.properties` (arquivo de configuração geral) e `connections.properties` (arquivos de configurações das localizações do repositório e fontes do *souce* e do *target*). A Figura 5-2 ilustra a tela principal da ferramenta, onde a partir dela podem ser realizadas as seguintes funcionalidades:

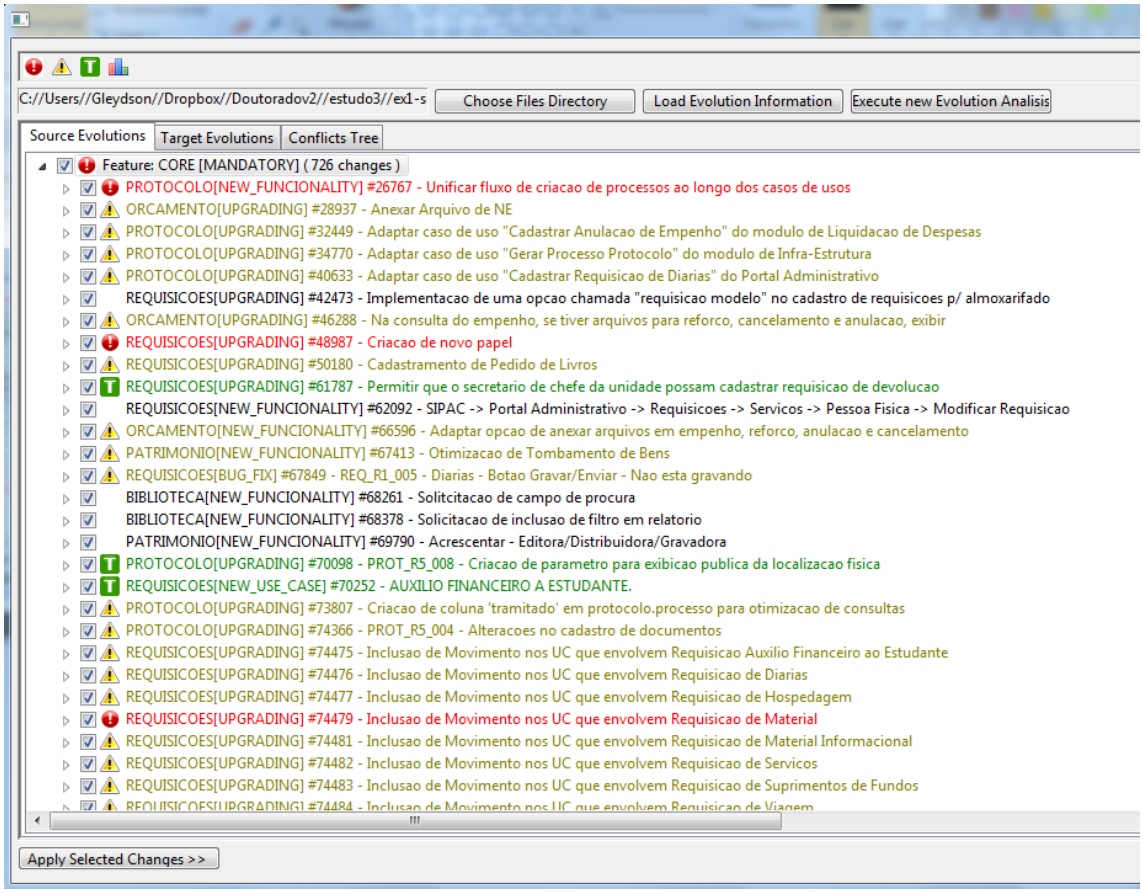


Figura 5-2 Tela principal do *plugin* de apoio à abordagem

1. ***Choose Files Directory*:** escolhe o diretório de trabalho principal da ferramenta. Não confundir com o *workspace*, pois como o *plugin* executa na ferramenta Eclipse, ele utiliza o *workspace* em uso pelo usuário. O diretório de trabalho é o local onde a ferramenta irá recuperar e gravar os arquivos necessários para a análise.

2. ***Load Evolution Information*:** carrega os arquivos XML de evoluções localizados no diretório de trabalho. São eles: `changeLogHistorySource.xml` e `changeLogHistoryTarget.xml`. Após carregá-los, a ferramenta os exibe no painel de visualização.

3. **Botões de análise de conflitos:** executa as análises de conflitos diretos () , indiretos () e textuais ().

4. **Geração de estatísticas:** botão () responsável pela geração das estatísticas de conflitos entre as versões em análise.

5. O botão ***Execute New Evolution Analisys*** é usado apenas para testes para simulação de conflitos sob de dados de teste.

5.2 Arquivos de Evoluções e Minerações

A geração do arquivo de evoluções foi realizada em ferramentas existentes de desenvolvimento de *software*, sendo desenvolvidos conectores de extração para o SIGProject, iProject e Redmine. A forma padrão de obtenção destes dados foi através da criação de um serviço REST em cada uma das ferramentas para geração dos dados em formato XML que podem ser consumidos diretamente através de uma URL de acesso ou importados através de um arquivo XML com os dados da evolução. Estes arquivos recebem o padrão de buildInformation_ [SYSTEM]_[VERSAO_INICIAL]_[VERSAO_FINAL], por exemplo: buildInformation_SIPAC_4.4.19_4.7.0.xml. Para sua implementação foram utilizadas

as tecnologias Jersey[3] para utilização do serviço REST e o JAXB para geração do formato XML.

A execução da mineração tem como entrada os arquivos de evoluções gerados pelo sistema de gestão de mudanças (SIGProject, Redmine, dentre outros) e sistema de controle de versões (SVN, neste caso), para então gerar um modelo com as informações relevantes sobre a evolução da linha de produto de *software* analisada. Este modelo, denominado de Modelo de Evoluções Mineradas, é utilizado pelo analisador de conflitos e *merge* de código.

A Figura 5-3 ilustra o modelo gerado como resultado da mineração. A classe `ChangeLogHistory` representa a classe principal para auxiliar na consulta dos resultados da mineração. Ela contém o nome do sistema e as versões inicial e final das versões mineradas. Cada instância da classe `ChangeLogHistory` contém N referências para objetos do tipo `ChangeLog`. Este último representa cada evolução individual realizada identificada por seu tipo (referência à classe `ChangeLogType`), sua versão e informações de identificação. O modelo prevê que uma mudança possua referencias às características afetadas do sistema (`FeatureChangeLog`), porém nas linhas analisadas, as ferramentas de desenvolvimento não continham tal informação. Cada classe `ChangeLog` possui N `ClassChangeLog`, as quais representam as classes evoluídas. Cada `ClassChangeLog`, por sua vez, possui uma coleção de `FieldChangeLog` e outra de `MethodChangeLog`, assim como listas de `ExtensionChangeLog`, `ImplementsChangeLog` e `AnnotationChangeLog`. Todas essas classes são usadas para armazenar evoluções específicas que ocorrem em elementos da linguagem Java. Cada `FieldChangeLog` e `MethodChangeLog` pode ter um conjunto de `AnnotationChangeLog`, que são evoluções em anotações Java.

[3] https://jersey.java.net/

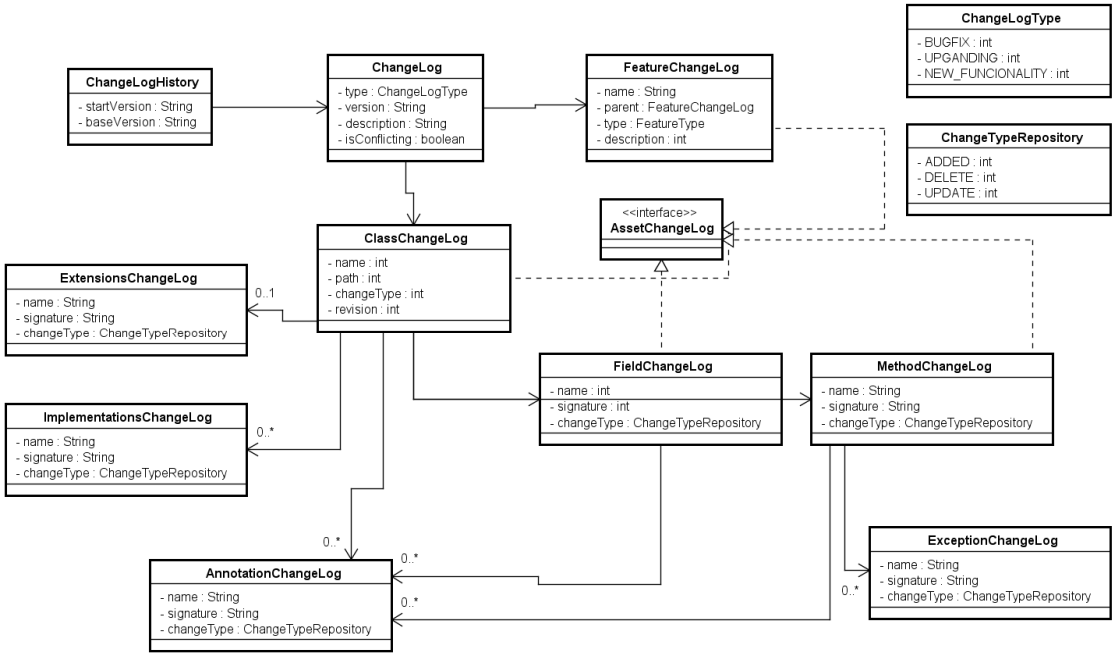


Figura 5-3 Modelo de Evoluções Mineradas

O modelo de evoluções mineradas, após ser minerado e construído, é armazenado no arquivo denominado `historyChangeLog[Source | Target].xml`. O painel de exibição da ferramenta, ilustrado na Figura 5-4, exibe este arquivo enumerando todas as evoluções atômicas obtidas a partir do processo de mineração.

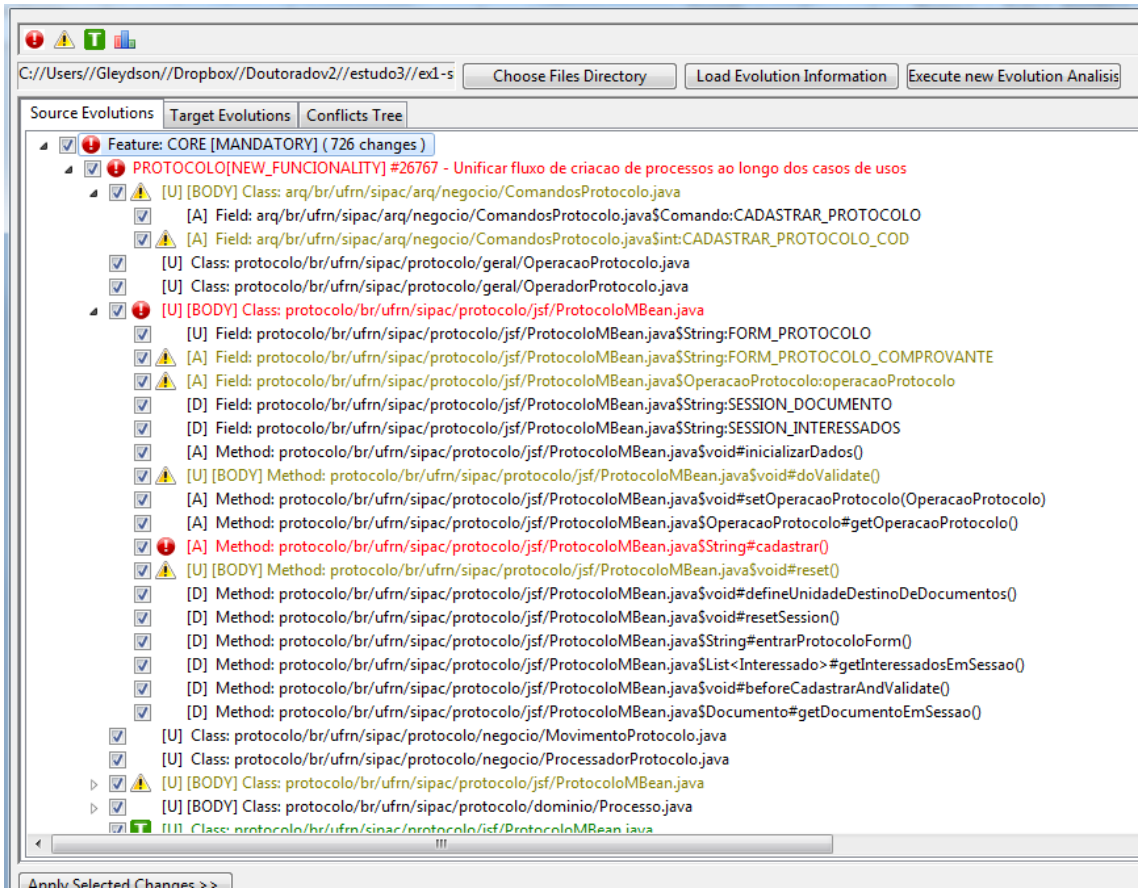


Figura 5-4 Exibição das minerações por mudança

5.3 Componente de Detecção de conflitos

O componente de detecção de conflitos é formado por três sub-pacotes: (i) analisador de conflitos diretos, (ii) analisador de conflitos indiretos e (iii) analisador de conflitos textuais. A Figura 5-5 ilustra as classes que implementam tais subsistemas. A interface `DirectConflictsDetectionStrategy` define o comportamento padrão para localização dos conflitos diretos. Ela é implementada pela classe `DirectConflictsDetectionEvolutionLimitedStrategy`, busca os conflitos diretos nas várias operações atômicas do *source* e do *target* e, ao encontrá-los, marca-os como conflitantes e os inclui no modelo de conflitos (*Conflict Model*).

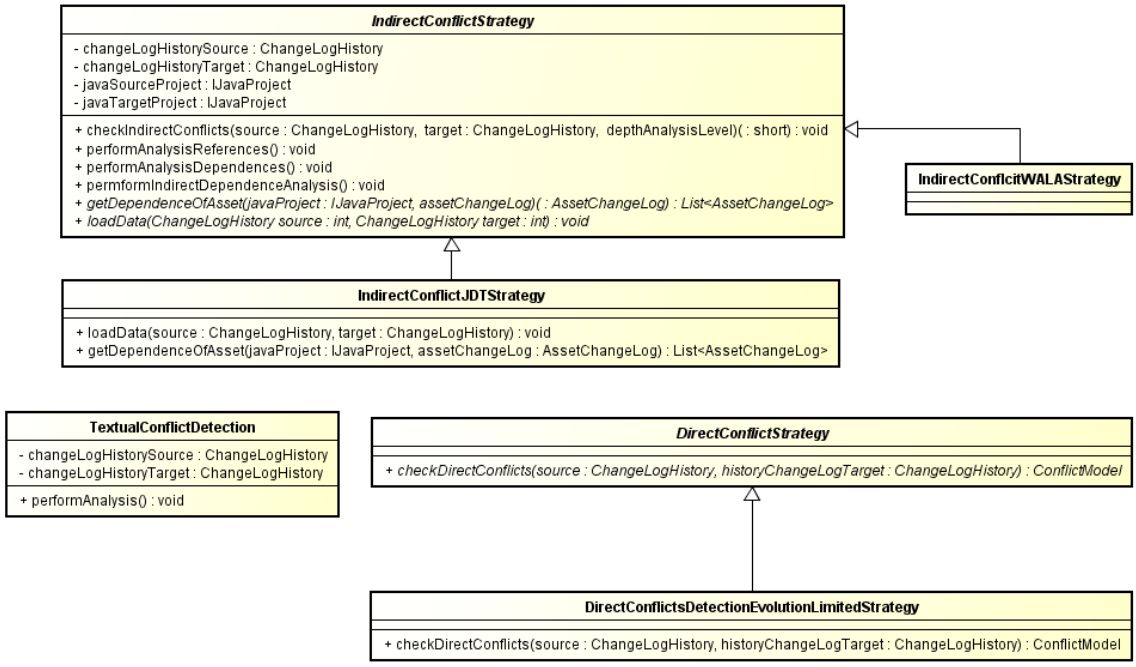


Figura 5-5 Interfaces e Classes de Análise de Conflitos

A classe `IndirectConflictsDetectionStrategy` implementa a estratégia de busca de conflitos indiretos e delega para as suas subclasses a implementação para a busca dos artefatos dependentes no método `getDependenceOfAsset()`. Neste caso, é possível que exista mais de uma implementação utilizando diferentes tecnologias de manipulação dos artefatos Java. A implementação atual da abordagem utiliza o JDT (*Java Development Tools*) do Eclipse, para manipular as interfaces e classes Java implementadas e realizar buscas específicas de dependências entre métodos de classes dentro do grafo de chamadas de interesse. Tal implementação está disponível na classe `IndirectConflictsDetectionJDTStrategy`.

A classe `TextualConflictsDetection` é ativada após a execução da análise de conflitos diretos e indiretos. Ela compara os artefatos do *source* e do *target,* e então verifica quais deles possuem modificações concorrentes sobre uma mesma classe, mas que não geraram conflitos diretos, sendo caracterizados assim como conflitos textuais.

As classes de análise de conflitos são responsáveis pela construção do modelo de conflitos (*Conflict Model*). A classe `ConflictModel` possui uma lista de conflitos do tipo `Conflict`. Cada `Conflict` possui a evolução do *source* (`assetEvolutionSource`) e do *target* (`assetEvolutionTarget`), além do seu tipo (`AssetConflictType`), conforme ilustra a Figura 5-6. Este modelo é armazenado em arquivos XML.

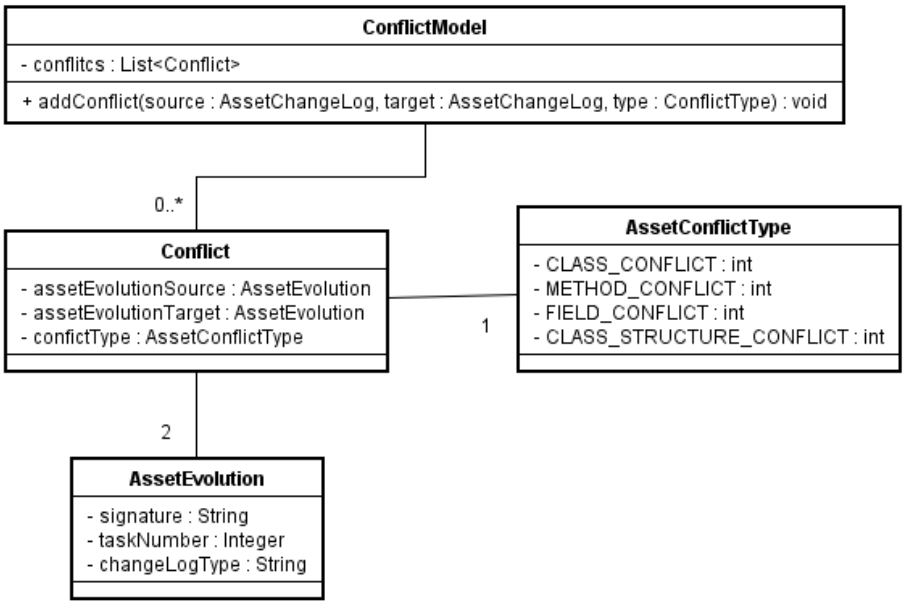


Figura 5-6 Modelo de Conflitos (*Conflict Model*)

5.4 Arquivo de Mapeamento de Variabilidades

O arquivo de mapeamento de variabilidades funciona como um repositório das variabilidades da linha de produto, indicando o seu tipo e localização no código fonte, ou seja, os artefatos impactados. A utilização das informações disponíveis em tal arquivo auxilia o analisador de conflitos a identificar se a evolução, em estado de conflito ou não, está acontecendo no núcleo ou nas variabilidades da LPS. A classe `VariabilityRepository` é a classe principal deste componente: a partir dela é possível realizar uma busca no modelo de variabilidades ilustrado na Figura 5-7. Cada variabilidade é representada pela classe `Variability`, que possui uma coleção indicando as suas localizações no código fonte, os `variantPoint`, e os valores possíveis que essa variabilidade pode assumir representada pela classe `Variant`. Cada variabilidade pode ter sua implementação do tipo `PATTERN` ou `CONDICIONAL_EXEC` (classe `VariabilityType`) e o tipo da característica que ela implementa, representada pela classe `FeatureType`. O arquivo XML, ilustrado na Figura 5-7, é o formato de armazenamento das variabilidades que é transformado no modelo de classes da Figura 5-7 através da tecnologia JAXB.

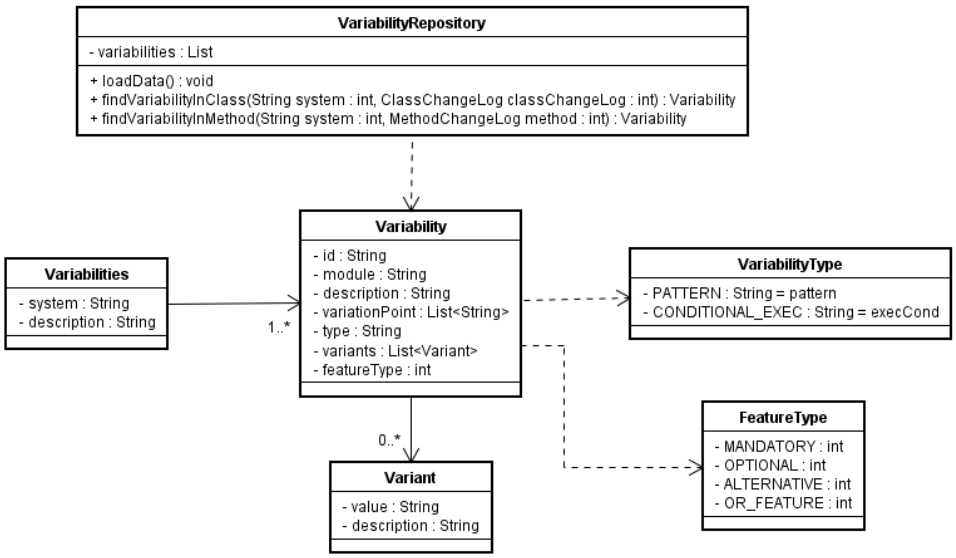


Figura 5-7 Modelo de variabilidades

```
...
<variability>
              <id>sigaa-ensino-01</id>
              <description>Estratégia para geração de matrículas de
alunos</description>

              <module>ensino</module>
              <type>plugin</type>
              <subtype>strategy</subtype>
              <featureType>alternative</subtype>

       <variationPoint>br.ufrn.sigaa.ensino.negocio.geracao_matricula.EstrategiaComposic
aoMatricula</variationPoint>

              <variants>
                     <description>Por ano e nível</description>

       <value>br.ufrn.sigaa.ensino.negocio.geracao_matricula.ComposicaoMatriculaPorAnoNi
vel</value>
              </variants>

              <variants>
                     <description>Por ano/periodo</description>

       <value>br.ufrn.sigaa.ensino.negocio.geracao_matricula.ComposicaoMatriculaPorAnoPe
riodo</value>
              </variants>
</variability>
.....
```

Figura 5-8 Arquivo XML com a base de variabilidades

5.5 Componente de Geração de Estatísticas

O componente de geração de estatísticas tem a função de percorrer os modelos de evoluções minerados, após a análise dos conflitos, gerar dados sintéticos e analíticos do *source* e do *target*. Os dados gerados detalham as evoluções e conflitos, exportando-os para arquivos de planilhas para permitir uma melhor análise. Os seguintes arquivos são gerados pela abordagem:

- ***Statistic_[source | target].properties:*** é um arquivo em formato propriedade=valor com estatísticas gerais, tais como: total de evoluções, totalizações por operações atômicas, total de evoluções por camada de *software*, total de evoluções conflitantes por tipo de conflito, dentre outras. É gerado um arquivo para o *source* e outro para o *target*.

- ***Evolutions_[source | target].csv:*** planilha contendo os dados de cada evolução atômica, indicando onde ela ocorre, qual artefato modifica, que conflito se manifestou, em que camada da arquitetura do *software* ela ocorre, a qual tarefa pertence e se ocorreu no núcleo ou variabilidade da LPS.

- **Task_[source | target].csv:** arquivo contendo as estatísticas de evoluções e conflitos sintetizadas por tarefa (*changelog*). Para cada tarefa o componente computa o total de conflitos textuais, diretos e indiretos, o tipo de tarefa (melhoria de funcionalidades, novos casos de usos e correção de *bugs*) e o total de conflitos ocorridos.

- **Conflicts.csv:** lista contendo todos os pares de conflitos calculados na comparação entre o *source* e o *target*.

5.6 Componente *Merge Engine*

O *merge engine* é o componente da abordagem responsável por aplicar as tarefas selecionadas do *source* no *target* e manter o código do *target* íntegro e compilável. O *merge engine* é ativado através do botão *Apply Selected Changes* da ferramenta

ilustrada na Figura 5-2. Sua implementação está disponível no pacote `br.ufrn.spl.ev.engines.merge`, o qual define os seguintes sub-pacotes:

- **br.ufrn.spl.ev.engines.merge.ast:** pacote contendo as classes de operações relativas à AST (*Abstract Syntax Tree*) para manipulação das classes Java.
- **br.ufrn.spl.ev.engines.merge.asset:** pacote contendo as classes *Field*, *Method*, *Type* e *TypeRefence* (*import*). Estas classes encapsulam os tipos do JDT (`org.eclipse.jdt.core.dom`) em classes da própria abordagem.
- **br.ufrn.spl.ev.engines.merge.engine:** pacote contendo as classes de execução do controle do *merge*, recuperando as classes da origem, chamando os demais componentes do *merge* e aplicando o código desejado para inserção no destino (*target*).

A classe `MergeCompilationUnit` é uma das principais classes do *merge engine*. Ela adiciona comportamentos de *merge* à classe padrão de Unidade de Compilação do JDT: `org.eclipse.jdt.core.dom.CompilationUnit`. Ela é construída a partir do objeto IType (que representa uma classe no JDT) e que constrói um *parser* AST para visitar todos os métodos, atributos, *source folder*, *imports* e então constrói a entidade `br.ufrn.spl.ev.engines.merge.asset.Type`. A classe `MergeCompilationUnit` implementa a interface `MergeOperations` que permite que a unidade de compilação possa ser manipulável através das operações atômicas.

As operações realizadas na AST são realizadas através classe `ASTRewrite` e `ListRewrite`, localizadas no pacote `org.eclipse.jdt.core.dom.rewrite`. A `ListRewrite` permite adicionar uma série de modificações na AST que são consolidadas de uma única vez através do uso da classe `TextEdit` (`org.eclipse.text.edits`) e `Document` (`org.eclipse.jface.text`).

A classe `br.ufrn.spl.ev.engines.merge.engine.MergeEngine` é a classe principal para ativação do *merge*. Ela executa o procedimento do *merge* através do método `execute()` ilustrado na Figura 5-9. Para cada *changelog* o componente registra o estado anterior e executa o *merge*.

```
	public void execute() throws Exception {

		for (FeatureChangeLog feature : historySource.getFeatures()) {
			for (ChangeLog changeLog : feature.getChangelogs()) {
				if (changeLog.isSelected()) {
					registerPreviousState(changeLog);
					executeChangeLog(changeLog);
				}
			}
		}

	}
```

Figura 5-9 Execução geral do *merge*

O método `registerPreviousState()` armazena o estado do código fonte de cada uma das classes envolvidas com o *merge* para ser usado na restauração em casos do *merge* conduzir a situações de falhas ou erros de compilação. O algoritmo geral de execução de *merge* pode ser ilustrado no código da Figura 5-10.

```
List<ClassChangeLog> relatedClasses = changelog.getClasses();

for (ClassChangeLog classe : relatedClasses ) {

	if (classe.getChangeType().equals(ChangeTypeRepository.ADDED) &&
classe.isSelected() ) {
		addClassInTarget(classe);
	}
	if (classe.getChangeType().equals(ChangeTypeRepository.DELETE) &&
classe.isSelected() ) {
	deleteClassInTarget(classe);
	}

	if (classe.getChangeType().equals(ChangeTypeRepository.UPDATED) &&
classe.isSelected() ) {
	    MergeCompilationUnit assetSource = new
	    MergeCompilationUnit(Repository.findClassInSource(classe.getClassName()));

	    MergeCompilationUnit assetTarget = new
MergeCompilationUnit(Repository.findClassInTarget(classe.getClassName()));

	for (MethodChangeLog method : classe.getMethods()) {

		Method metodoSource = assetSource.getMethod(method.getName());
		if (method.getChangeType().equals(ChangeTypeRepository.ADDED) &&
method.isSelected()) {
			assetTarget.addMethod(metodoSource);
		}

		if (method.getChangeType().equals(ChangeTypeRepository.DELETE) &&
method.isSelected()) {

			Method methodTarget = assetTarget.getMethod(method.getName());
			assetTarget.removeMethod(methodTarget);
		}
		if (method.getChangeType().equals(ChangeTypeRepository.UPDATED) &&
method.isSelected()) {
			assetTarget.updateMethod(metodoSource);
		}

	}
	for (FieldChangeLog field : classe.getFields()) {
```

```
                Field fieldSource = assetSource.getField(field.getName());

                if (field.getChangeType().equals(ChangeTypeRepository.ADDED) &&
field.isSelected() ) {
                        assetTarget.addField(fieldSource);
                }

                if (field.getChangeType().equals(ChangeTypeRepository.DELETE) &&
field.isSelected()) {

                        Field fieldTarget = assetTarget.getField(field.getName());

                        assetTarget.removeField(fieldTarget);
                }
                if (field.getChangeType().equals(ChangeTypeRepository.UPDATED) &&
field.isSelected()) {
                        assetTarget.updateField(fieldSource);
                }
        }
        assetTarget.applyChanges();
}
```

Figura 5-10 Código de controle do Merge Engine

Há outras classes utilitárias no *merge engine* que visam alertar o usuário para as operações a serem testadas, ou seja, aquelas operações do *target* que foram alvos de conflitos indiretos. Também classes utilitárias para utilização das funções do JDT no projeto.

5.7 Sumário

Este capítulo abordou os aspectos de implementação da ferramenta de suporte à abordagem proposta. Inicialmente abordou-se a arquitetura geral da implementação e os componentes que fazem parte de sua arquitetura. Em seguida, detalhou-se a construção do modelo de mineração, a análise de conflitos, o banco de variabilidades, o analisador de estatísticas e a execução do merge *engine*. O próximo capítulo detalha o estudo realizado para caracterizar os conflitos que ocorrem em produtos clonados de uma linha de produto de sistemas de informação web.

6 Estudo Empírico de Caracterização de Conflitos e Resolução de *Merges* de Linhas de Produto Clonadas

Este capítulo descreve um estudo empírico conduzido com o objetivo de caracterizar conflitos que ocorrem, durante os procedimentos envolvidos na integração de evoluções independentes de linhas de produto de *software* clonadas. Tal caracterização é útil e fundamental para entender o problema e conflitos da integração (*merge*) de linhas de produto de software clonadas, assim como demonstrar o potencial do uso da abordagem proposta nesta tese para lidar com tal problema. A Seção 6.1 apresenta os principais objetivos e as questões de pesquisa que foram investigadas no decorrer deste estudo. A Seção 6.2 detalha as linhas de produto de *software* analisadas, com seus respectivos tamanhos, quantidade e tipo de tarefas analisadas e tipos de evoluções nos artefatos de implementação. A Seção 6.3 detalha as fases para condução do estudo. A seção 6.4 detalha as métricas que foram coletadas no estudo. Finalmente, a Seção 6.5 apresenta e discute os resultados obtidos.

6.1 Objetivos e Questões de Pesquisa

Os principais objetivos do estudo empírico conduzido foram: (i) caracterizar que tipos de conflitos de código acontecem ao evoluir e integrar versões clonadas de linhas de produto de *software* de sistemas de informação web; (ii) caracterizar que tipos de tarefas são responsáveis por tais conflitos e onde acontecem (núcleo ou variabilidades da LPS, camada/modulo da arquitetura); e (iii) entender quais daqueles conflitos podem ser integrados de forma automatizada, semi-automatizada ou manual, através do uso de operações de *merge* de uma LPS clonada para a outra ou com o seu ancestral (*source*).

Em particular, busca-se responder às seguintes questões de pesquisa:

- QP1: Que tipos de conflitos de código ocorrem durante a evolução e *merge* das linhas de produtos de software clonadas?
- QP2: Que tipos de tarefas – correção de *bugs*, melhorias de funcionalidades, novos casos de usos – trazem mais conflitos no núcleo e variabilidade da linha de produto?
- QP3: Onde e como os conflitos ocorrem ao longo das camadas da LPS?
- QP4: Onde e como os conflitos ocorrem ao longo do núcleo e variabilidades da LPS?

6.2 Caracterização das Linhas de Produto Analisadas

Para conduzir o estudo foram selecionadas quatro evoluções concomitantes das linhas de produto SIPAC e SIGAA, onde cada evolução representa o desenvolvimento de duas linhas de produto clonadas, derivadas de uma mesma versão de desenvolvimento. A Tabela 6-1 enumera o total de tarefas de desenvolvimento que ocorreram para cada evolução de linhas de produto clonadas. As siglas E1, E2, E3 e E4 são referências ao número da evolução clonada analisada, onde E1 é um clone do sistema SIPAC, e E2, E3 e E4 são clones do sistema SIGAA. Por exemplo, a Tabela 6-1 mostra que a evolução E1 do sistema SIPAC teve 726 solicitações de mudanças durante 7 meses na instituição UFRN, mantenedora do *source*, e em paralelo, ocorreram 54 solicitações de mudança em outra instituição que utiliza a LPS clonada (*target*), durante o período de 6 meses. O estudo analisa os conflitos considerando que a reconciliação ocorre, em via de regra, na aplicação de tarefas da LPS *source* na LPS *target*.

As Tabelas 6-3 e 6-4 detalham as requisições de mudanças das evoluções do *source* e do *target*, respectivamente, por tipo. Os tipos caracterizados são: (i) novos casos de uso, (ii) melhoria de funcionalidades e (iii) correção de *bugs*. Por exemplo, das 726 evoluções do E1, 391 foram de correção de *bugs*, 310 de melhoria de funcionalidades e 25 de novos casos de uso.

As Tabela 6-4 e Tabela 6-5 detalham cada evolução clonada em termos de operações atômicas de evolução do *source* e do *target*, respectivamente. Observe que no *source* foram analisadas 18.724 operações atômicas, entre adição e remoção de

classes e suas atualizações: adição, remoção, atualização de métodos e adição, remoção e atualização de atributos. Já no *target* foram analisadas 2.986 operações atômicas, distribuídas conforme mostrado na Tabela 6-5.

	SIPAC		**SIGAA**					
Evolução Clonada	**E1**		**E2**		**E3**		**E4**	
	Tarefas	**Meses**	**Tarefas**	**Meses**	**Tarefas**	**Meses**	**Tarefas**	**Meses**
Source	726	7	238	9	809	11	809	11
Target	54	6	57	9	57	9	238	9
Total de evoluções analisadas	**780**	**-**	**295**	**-**	**866**	**-**	**1047**	**-**

Tabela 6-1 – Solicitações de mudanças realizadas por evolução paralela

Tipo de Tarefa	**E1**		**E2**		**E3 e E4**	
Correção de *bugs*	391	53,9%	169	71,0%	389	48,1%
Melhoria de Funcionalidades	310	42,7%	46	19,3%	393	48,6%
Novas Funcionalidades	25	3,4%	23	9,7%	27	3,3%
Total de Tarefas	726	100,0%	238	100,0%	809	100,0%

Tabela 6-2 Evoluções por Tipo - *Source*

Tipo de Tarefa	**E1**		**E2 e E3**		**E4**	
Correção de *bug*	35	64,8%	47	82,5%	169	71,0%
Melhoria de Funcionalidades	15	27,8%	8	14,0%	46	19,3%
Novas Funcionalidades	4	7,4%	2	3,5%	23	9,7%
Total de Tarefas	54	100,0%	57	100,0%	238	100,0%

Tabela 6-3 - Total de tarefas por Tipo - *Target*

Tipo de Evolução	**E1**		**E2**		**E3 e E4**		**Total**	**Média**
Class Add	45	0,8%	3	0,1%	188	1,8%	236	0,9%
Class Delete	11	0,2%	5	0,2%	85	0,8%	101	0,4%
Field Add	847	14,3%	221	10,1%	1061	10,0%	2129	11,5%
Field Delete	118	2,0%	34	1,5%	397	3,7%	549	2,4%
Field Update	131	2,2%	41	1,9%	426	4,0%	598	2,7%
Method Add	1750	29,5%	582	26,5%	2546	24,0%	4878	26,7%
Method Delete	237	4,0%	101	4,6%	1198	11,3%	1536	6,6%
Method Update	2785	47,0%	1216	55,4%	4696	44,3%	8697	48,9%
Total	5924	100,0%	2203	100,4%	10597	100%	**18724**	100,0%

Tabela 6-4 - Total de evoluções atômicas por evolução clonada – *Source*

Tipo de Evolução	E1		E2 e E3		E4		Total	Média
Class Add	0	0,00%	2	0,5%	3	0,0%	5	0,17%
Class Delete	0	0,00%	0	0,0%	5	0,0%	5	0,17%
Field Add	38	10,73%	77	17,9%	221	2,1%	336	11,25%
Field Delete	2	0,56%	1	0,2%	34	0,3%	37	1,24%
Field Update	7	1,98%	9	2,1%	41	0,4%	57	1,91%
Method Add	81	22,88%	140	32,6%	582	5,5%	803	26,89%
Method Delete	9	2,54%	6	1,4%	101	1,0%	116	3,88%
Method Update	217	61,30%	194	45,2%	1216	11,5%	1627	54,49%
Total	354	100,00%	429	100,0%	2203	100%	**2986**	100,00%

Tabela 6-5 Total de evoluções atômicas por evolução clonada - *Target*

6.3 Fases do Estudo

Para responder às questões de pesquisa definidas na seção 6.1, o estudo foi organizado nas seguintes fases:

1. **Escolha das versões de evolução das LPS clonadas (*source* e *target*) a serem analisadas no estudo:** As linhas de produto de software SIPAC e SIGAA foram escolhidas pelo interesse dos pesquisadores envolvidos nesta pesquisa com os projetos desenvolvidos, assim como por se tratarem de LPS Web de larga escala desenvolvidas usando boas práticas de desenvolvimento (padrões arquiteturais e de projeto, tecnologias modernas) adotadas pela comunidade. Outro motivo para sua escolha foi a ausência de LPS de sistemas de informações Web de código aberto, onde fosse possível o acesso ao ambiente de desenvolvimento para analisar suas evoluções em detalhes e alterar, quando fosse o caso, as ferramentas envolvidas no seu desenvolvimento. No caso em particular, havia flexibilidade para acesso aos repositórios de software dos sistemas de controle de versão e gerência de mudanças dos projetos, além da proximidade com as equipes de desenvolvimento de cada uma das LPS clonadas. No que tange a escolha das evoluções de LPS clonadas, o critério utilizado exigiu que as versões da LPS *source* e no *target,* possuíssem tarefas de todos os tipos consideradas nesse estudo: melhoria de funcionalidades, correção de *bugs* e novos casos de usos.

2. **Extração dos relatórios de versão (*changelogs*) dos sistemas de gestão de mudanças (iProject e SIGProject) do *source* e do *target*:** Para cada um dos projetos selecionados utilizou-se de sua ferramenta de gestão de mudanças para extração dos dados de evolução das tarefas de desenvolvimento (*issues*) e respectivas revisões em classes/interfaces no sistema. Foram também gerados arquivos XML contendo todas as evoluções no formato utilizado pela abordagem.

3. **Mineração do repositório de código (SVN) para obtenção das operações atômicas de evolução para cada tarefa (*task changelog*):** A partir dos arquivos obtidos no passo anterior, identificou-se cada publicação no repositório (*commit*) e suas associações com as tarefas. Para cada *commit*, a mineração extrai as classes e interfaces modificadas, identifica os elementos de cada classe (atributos, métodos, dentre outros) que foram criados, alterados ou removidos. Finalmente, tais informações mineradas são inseridas no arquivo XML gerado no passo anterior (Figura 5-3).

4. **Execução da ferramenta da abordagem para detecção dos conflitos diretos, indiretos e textuais:** De posse da informações das evoluções das LPS clonadas em termos de tarefas realizadas e modificações atômicas nas classes para cada uma das tarefas, a ferramenta da abordagem realizou a análise de conflitos textuais, diretos e indiretos. A análise dos conflitos diretos e textuais foi realizada processando as informações extraídas através da mineração de repositórios. Já a análise de conflitos indiretos faz uso de análise estática de código para calcular as dependências entre elementos (métodos, atributos) do código fonte durante a execução. Por este motivo, a análise dos conflitos indiretos exige um esforço computacional diretamente proporcional a quantidade de classes e dependências entre os elementos de tal classe. Já as análises dos conflitos diretos e textuais pode ser realizada com tempo de execução bastante reduzido em comparação a análise de conflitos indiretos.

5. **Análise das evoluções e conflitos ocorridos no *source* e no *target* para cálculo e geração do arquivo de estatísticas de interesse do estudo:** Após o cálculo dos conflitos e a geração do *conflict model,* a abordagem habilita a geração das estatísticas globais e detalhadas, que representam as métricas consideradas no estudo. As estatísticas globais totalizam os dados por versão, por exemplo: total de evoluções atômicas da versão, total de conflitos diretos, total de conflitos indiretos, dentre outros. Já as estatísticas detalhadas geram dados de modificações atômicas e conflitos por evolução (arquivo de evoluções) e por tarefa (arquivo de tarefas).

6. **Análise quantitativa e qualitativa dos resultados obtidos para o estudo de forma a buscar respostas para as perguntas de pesquisa:** de posse dos resultados das métricas de interesse do estudo, realizou-se a análise das questões de pesquisa. Em seguida, após análise dos resultados obtidos, foram realizadas análises qualitativas do código-fonte de forma a verificar as razões para parte dos resultados obtidos.

6.4 Questões de Pesquisa e Métricas do Estudo

Para responder a cada uma das questões de pesquisas do estudo empírico foi preciso definir métricas específicas relacionadas à quantificação de conflitos encontrados nas evoluções clonadas avaliadas. As Tabelas 6-6, 6-7, 6-8 e 6-9 enumeram as questões e sub-questões de pesquisas, assim como as métricas utilizadas em cada uma delas.

QP1: Quais os tipos e quantidade de conflitos de código que ocorrem durante a evolução e *merge* de linhas de produto Web clonadas?
QP-1.1 - Quantos e quais tipos de conflitos ocorrem na reconciliação das linhas de produto *core* e *target*?
Métrica: Número de conflitos diretos, indiretos e textuais que ocorrem durante a reconciliação (*merge*) das duas linhas de produto de software clonadas.

QP-1.2 - Quantas e quais tarefas do *source* e do *target* apresentam conflitos durante a reconciliação das linhas de produto clonadas?
Métricas: (i) Número de tarefas do *source* e do *target* com e sem conflitos; (ii) Número de tarefas que possuem uma determinada combinação de conflito (apenas textual, direto e indireto; ou conflitos textual e direto; e as demais combinações)
QP-1.3 - Qual a frequência de conflitos que ocorrem por tarefa no *source* e no *target ?*
Métrica: Quantidade de tarefas que apresentam 1, 2, …, N-1 e N conflitos no *source* e no *target*.
QP-1.4 - Quantos e quais conflitos acontecem em cada operação atômica no *source* e no *target?*
Métrica: Número de operações atômicas do *source* e do *target* nos quais ocorreu conflitos diretos, indiretos e textuais.
QP-1.5 - Qual a distribuição dos conflitos nas diversas operações atômicas?
Métrica: Número de conflitos diretos e indiretos para cada tipo de operação atômica (adicionar, remover e alterar atributos e métodos) do *source* e do *target*
QP-1.6 - Qual a profundidade média no grafo de chamados na ocorrência de conflitos indiretos?
Métrica: Percentual de conflitos indiretos ocorridos nas profundidades 1,2,3,4,5 e 6.

Tabela 6-6 Métricas da questão de pesquisa 2

QP2: Que tipos de tarefas – correção de *bugs*, melhoria de funcionalidades, novos casos de usos – trazem mais conflitos no núcleo e variabilidade da linha de produto?
QP-2.1 - Como os conflitos se manifestam nos diferentes tipos de tarefas no *source* e no *target*?
Métrica: Números de conflitos diretos, indiretos e textuais ocorridos em tarefas de correção de *bugs*, melhoria de funcionalidades e novos casos de usos.
QP-2.2 - Qual o percentual de manifestação das diferentes combinações de conflitos nas tarefas de correção de *bugs*, novas funcionalidades e aprimoramentos?
Métrica: Percentual de conflitos em cada tipo de tarefa que possuem uma determinada combinação de conflito (apenas textual, direto e indireto; ou conflitos textual e direto; e as demais combinações)

Tabela 6-7 Métricas da questão de pesquisa 2

QP3: Onde e como os conflitos ocorrem ao longo das camadas da LPS?
QP-3.1 - Em que camadas/módulos da LPS, os conflitos de código ocorrem na LPS, tanto no *source* quanto no *target*?
Métrica: Número de conflitos diretos, indiretos e textuais para cada camada do *source* e do *target*.
QP-3.2 – Como cada tipo de tarefa do *source* (resolução de *bugs*, melhoria de funcionalidades e novos casos de usos) conflita ao longo das camadas?
Métricas: Quantidade de conflitos que tarefas do *source* conflitam com outras tarefas do *target* (conflito em 1, 2, 3, 4 ou todas as camadas)

Tabela 6-8 Métricas da questão de pesquisa 3

QP4: Onde e como os conflitos ocorrem ao longo do núcleo e variabilidades da LPS?
QP-4.1 – Em que parte da LPS -núcleo ou variabilidades - ocorrem as evoluções e os conflitos em operações atômicas?
Métrica: Números de evoluções, tarefas e conflitos relacionados com *source* e *target.*
QP-4.2 – Que tipos de conflitos ocorrem no núcleo ou nas variabilidades da LPS?
Métricas: Números de conflitos (direto, indireto e textual) que ocorrem no núcleo e nas variabilidades das LPS clonadas.
QP-4.3 - No que diz respeito aos conflitos que ocorreram em variabilidades, onde foi maior a ocorrência em técnicas composicionais (padrões de projeto) ou anotativas (execução condicional)?
Métricas: Números de conflitos que ocorrem em implementações das variabilidades das LPS clonadas usando técnicas composicionais (padrões de projeto) ou anotativa (execução condicional).

Tabela 6-9 Métricas da questão de pesquisa 4

6.5 Análise dos Resultados

Esta seção apresenta e discute os resultados obtidos para o estudo, em termos das diferentes questões de pesquisa definidas.

6.5.1 QP1 – Que tipos de conflitos de código ocorrem durante a evolução e *merge* das linhas de produtos de software clonadas?

O objetivo desta questão de pesquisa QP1 é caracterizar que tipos de conflitos ocorrem na reconciliação de linhas de produto de sistemas de informação Web clonadas e que tiveram evoluções concomitantes. Os conflitos são analisados considerando quatro sub-análises e seis sub-questões de pesquisa: (i) considerando o total de conflitos manifestados – QP-1.1; (ii) a análise por tarefa (*changelog*) – QP-1.2 e QP-1.3 e (iii) a análise por modificação atômica (evoluções obtidas a partir da mineração) – QP-1.4 e QP1.5 e (iv) a análise da profundidade dos conflitos indiretos – QP-1.6. O conflito em uma tarefa é aquele em que pelo menos uma de suas operações atômicas de evolução do *source* conflitou com outra operação atômica do *target*. Dessa forma, podemos ter tarefas com poucos ou muitos conflitos, dependendo do total de operações atômicas conflitantes.

6.5.1.1 QP-1.1 - Quantos conflitos e quais tipos ocorrem na reconciliação das linhas de produto core e target?

Esta análise busca identificar o total de conflitos manifestados em cada evolução clonada. Cada conflito representa um par de operações atômicas incompatíveis entre tarefas do *source* e do *target,* respectivamente. Observe que cada conflito é calculado através da análise individualizada do produto cartesiano das operações atômicas do *source* e do *target*. A Tabela 6-10 enumera o resultado obtido.

Tipo de Conflito	E1	E2	E3	E4	Total	%
Conflitos Diretos	64	4	14	120	202	3,3%
Conflitos Indiretos Referências (*Target-Source*)	65	75	607	1.646	2.393	39,2%
Conflitos Indiretos Dependências (*Source-Target*)	113	46	262	2.480	2.901	47,5%
Conflitos Textuais	125	31	89	371	616	10,1%
Totais	**367**	**156**	**972**	**4.617**	**6.112**	**100,0%**

Tabela 6-10 Total de Conflitos

Observe que a maioria dos conflitos manifestados são indiretos, representando 86,7% (39,2 + 47,5). Isto significa que boa parte dos potenciais conflitos na reconciliação de versões não são aparentes, pois são manifestados nas dependências e não diretamente nos artefatos. Os conflitos indiretos dependências (*source-target*) é o mais frequente em média, significando que mudanças em artefatos do *target* que são chamados por artefatos modificados por tarefas do *source* ocorre de maneira mais frequente.

Para se ter uma base comparativa do total de conflitos pode-se comparar com o total de evoluções atômicas (mudanças mineradas nos artefatos) realizadas no clone (*target*). A Tabela 6-11 enumera esta média de conflitos considerando o total de conflitos e quantas evoluções atômicas cada clone teve. Na evolução clonada E1 da linha de produto SIPAC, por exemplo, foi gerado 1,04 conflito para cada evolução atômica no *target* (clone). As evoluções clonadas E3 e E4 mantiveram uma média de aproximadamente 2 conflitos gerados para cada evolução atômica. Já a evolução E2, manifestou menos conflitos, sendo um total de 0,36 conflitos por evolução atômica.

Observa-se que a média de E3 e E4 foi aproximada. Este caso é interessante para análise pois representam dois clones da mesma LPS *source* para diferentes clientes-instituições que sofreram mudanças para diferentes contextos de negócio do módulo de graduação, um dos maiores do sistema SIGAA, e mesmo assim, manifestaram uma média similar de conflitos. Já a análise E2 mostra que a reconciliação dos clones de E4 como *source* e E3 como *target* manifesta menos conflitos do que se compará-los com os seus respectivos *sources* da clonagem.

Evolução Clonada	**E1**	**E2**	**E3**	**E4**
Conflitos	367	156	972	4617
Evoluções atômicas no Clone (*target*)	354	429	429	2203
Média	1,04	0,36	2,27	2,10

Tabela 6-11 Média de Conflitos por Evolução atômica no clone (*target*)

Quanto maior a quantidade de modificações em atributos/métodos (operações atômicas) ocorrida na linha de produto clonada (*target*), para um mesmo *source*, maior a expectativa do número de conflitos, uma vez que mais artefatos serão modificados, aumentando potencialmente a probabilidade de conflitos. No caso específico do nosso estudo, as evoluções E3 e E4 mostraram que tal aumento no número de evoluções no *source* pode ocasionar um aumento de conflitos no *target*, conforme mostra a Tabela 6-10 onde a partir da mesma LPS *source* (evoluções clonadas E3 e E4), o número de conflitos cresceu de forma quase linear. Entretanto, outros fatores podem também influenciar em tal aumento, como por exemplo os tipos de tarefas, conforme será explorado nas demais questões de pesquisa. Adicionalmente, por se tratar de um cenário localizado, não é possível generalizar tal comportamento, apenas auferir que a expectativa foi confirmada nos exemplos testados.

Resolução automática de Conflitos. Pelos números apresentados na Tabela 6-10 verifica-se a contribuição da abordagem proposta nesta tese na resolução de conflitos: (i) 10,1% dos conflitos encontrados são textuais e podem, portanto, ser resolvidos de forma automatizada pela abordagem; (ii) 86,7% dos conflitos são indiretos e podem ser resolvidos de forma semi-automatizada; e (iii) apenas 3,3% dos conflitos são diretos e precisam ser resolvidos de forma manual pelo usuário. No geral, 96,8% dos conflitos encontrados podem ser resolvidos de forma automatizada ou semi-automatizada.

6.5.1.2 QP-1.2 - Quantas e quais tarefas do source e do target apresentam conflitos durante a reconciliação das linhas de produto clonadas?

A análise do total de conflitos manifestados e sua relação com as operações atômicas, foco da sub-questão QP-1.1, é importante uma vez que para resolver os conflitos é necessário analisar métodos e atributos diretamente, já que é o nível mais elementar da análise. No entanto, em geral, o interesse de reconciliação é no contexto de tarefa, por exemplo: recuperar um novo caso de uso do *source*, aplicar as tarefas de resolução de *bugs*, dentre outras. Um conflito, por menor que seja, demanda uma atenção à tarefa e pode acarretar atrasos no processo de *merge* das linhas de produtos clonadas.

Evolução Clonada	Total de Conflitos	Tarefas do *Source*	Tarefas do *Target*	Média de Conflitos por Tarefa do *Target*
E1	367	726	**54**	6,80
E2	156	238	**57**	2,74
E3	972	809	**57**	17,05
E4	4617	809	**238**	19,40

Tabela 6-12 Total de Conflitos por Evolução

A Tabela 6-12 enumera o total de conflitos em cada evolução clonada comparando-os com o total de evoluções (tarefas) do *clone*. Observe que a evolução clonada E1 (sistema SIPAC) manifestou 6,8 conflitos em média, por tarefa. A evolução E2, que é o comparativo entre os clones de E3 e E4, manifestou 2,74 conflitos por tarefa. Já as evoluções clonadas E3 e E4 tiveram médias aproximadas (17,05 e 19,04), assim como aconteceu para estas duas evoluções na média por operação atômica.

LPS	Total de Tarefas	Tarefas com Conflitos	Média
Source	2582	742	28,7%
Target	406	282	69,5%

Tabela 6-13 Resumo das Tarefas dos Conflitos no Source e Target

Observe, conforme ilustrado na Tabela 6-13, que a ocorrência de tarefas com conflitos no produto clonado (*target*) é 2,4 vezes superior ao *source*, com 69,5% contra 28,7%. Este resultado é esperado uma vez que o *target* é o produto clonado adaptado. Já o *source*, além de possuir mais tarefas e evoluções, contém também diversas tarefas de outros módulos que não sofreram adaptações no produto clonado, não manifestando conflitos para sua reconciliação.

Uma tarefa pode conter os três tipos de conflitos a depender da ocorrência em suas operações atômicas. Desta forma, foi realizada uma análise das possíveis combinações de conflito da seguinte forma: (i) tarefas somente com conflitos textuais; (ii) somente com conflitos diretos; (iii) somente com conflitos indiretos; (iv) com conflitos diretos e indiretos; (v) conflitos diretos e textuais; (vi) conflitos indiretos e textuais; e (vii) com todos os conflitos. As Tabelas Tabela 6-14 e Tabela 6-15 enumeram a distribuição dos percentuais encontrados no *source* e no *target*, e as Figuras 6-1 e 6-2 ilustram graficamente estes percentuais.

	E1		**E2**		**E3**		**E4**		**Média**
Total de Tarefas	726		238		809		809		
Sem conflitos	559	77,0%	190	79,8%	636	78,6%	455	56,2%	71,3%
Total com Conflitos	**167**	**23,0%**	**48**	**20,2%**	**173**	**21,4%**	**354**	**43,8%**	**28,7%**
Distribuição dos Conflitos no Source (100¨% dos conflitos)									
Conflitos Textuais	65	9,0%	18	7,6%	43	5,3%	103	12,7%	**30,9%**
Conflitos Diretos	29	4,0%	2	0,8%	5	0,6%	20	2,5%	7,5%
Conflitos Indiretos	32	4,4%	17	7,1%	85	10,5%	86	10,6%	**29,6%**
Conflitos Diretos e Indiretos	6	0,8%	1	0,4%	3	0,4%	13	1,6%	3,1%
Conflitos Diretos e Textuais	3	0,4%	0	0,0%	1	0,1%	5	0,6%	1,2%
Conflitos Indiretos e Textuais	21	2,9%	9	3,8%	33	4,1%	94	11,6%	21,2%
Conflitos Textuais, Diretos e Indiretos	11	1,5%	1	0,4%	3	0,4%	33	4,1%	6,5%

Tabela 6-14 Conflitos por Tarefa - *Source*

	E1		E2		E3		E4		%
Total de Tarefas	54		57		57		238		
Sem conflitos	17	31,5%	24	42,1%	20	35,1%	63	26,5%	30,5%
Total com Conflitos	**37**	**68,5%**	**33**	**57,9%**	**37**	**64,9%**	**175**	**73,5%**	**69,5%**
Distribuição dos Conflitos no Target (100¨% dos conflitos)									
Conflitos Textuais	2	3,7%	4	7,0%	2	3,5%	10	4,2%	6,4%
Conflitos Diretos	4	7,4%	1	1,8%	0	0,0%	4	1,7%	3,2%
Conflitos Indiretos	10	18,5%	20	35,1%	16	28,1%	93	39,1%	**49,3%**
Conflitos Diretos e Indiretos	3	5,6%	1	1,8%	1	1,8%	15	6,3%	7,1%
Conflitos Diretos e Textuais	4	7,4%	1	1,8%	1	1,8%	1	0,4%	2,5%
Conflitos Indiretos e Textuais	6	11,1%	6	10,5%	9	15,8%	26	10,9%	16,7%
Conflitos Textuais, Diretos e Indiretos	8	14,8%	0	0,0%	8	14,0%	26	10,9%	14,9%

Tabela 6-15 Conflitos por Tarefa – *Target*

Dentre as tarefas com conflitos, que representam 28,7% do *source* e 69,5% do *target* a distribuição dos conflitos é a seguinte: 30,9% das tarefas do *source* e em 6,4% do *target* tiveram somente conflitos textuais. Em 7,5% das tarefas do *source* e em 3,2% das tarefas do *target* ocorreram conflitos diretos. Já a ocorrência de tarefas com somente conflitos indiretos ocorreu em 29,6% do *source* e em 49,3% das tarefas do *target*. Desta forma, em uma análise por tipo de conflito: (i) 68% dos conflitos em tarefas *source* e 58,9% do *target* são apenas de um tipo de conflito; (ii) 25,5% do *source* e 26,2% do *target* possuem dois tipos de conflitos; e (iii) 6,5% do *source* e 14,9% do *target* todos os conflitos.

Se considerarmos os percentuais de ocorrência de tarefas que tem pelo menos um dos tipos de conflitos, enumerados na Tabela 6-16 verifica-se que o *source* possui maiores incidências de conflitos textuais e indiretos (59,7% e 60,4%) e o *target* de conflitos indiretos (87,9%). Observe que se somarmos os percentuais eles poderão ultrapassar os 100% pelo fato de que os diferentes tipos de conflitos podem ocorrer simultaneamente em uma mesma tarefa.

Tipo de Conflito	*Source*	*Target*
Tarefas com Conflitos Textuais	59,7%	40,4%
Tarefas com Conflitos Diretos	18,3%	27,7%
Tarefas com Conflitos Indiretos	60,4%	87,9%

Tabela 6-16 Percentual de Conflitos por Tarefa

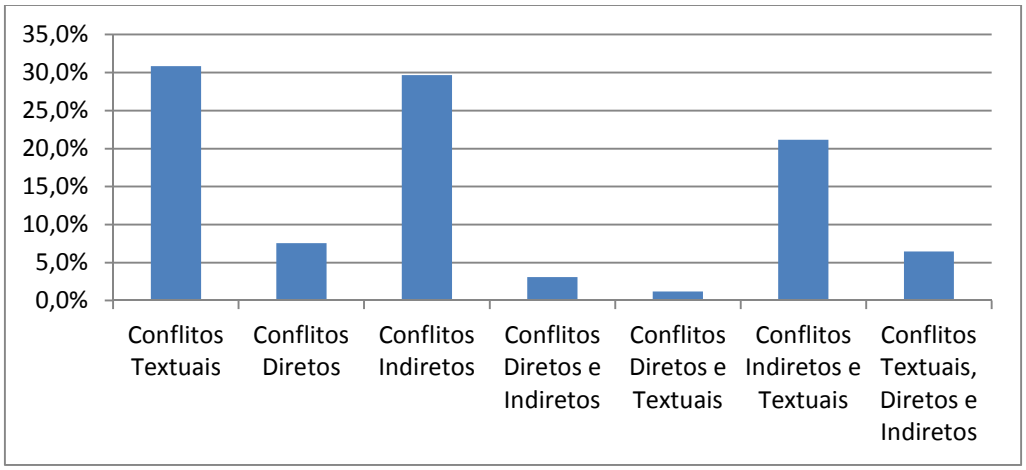


Figura 6-1 Percentual de Conflitos em Tarefas do *Source*

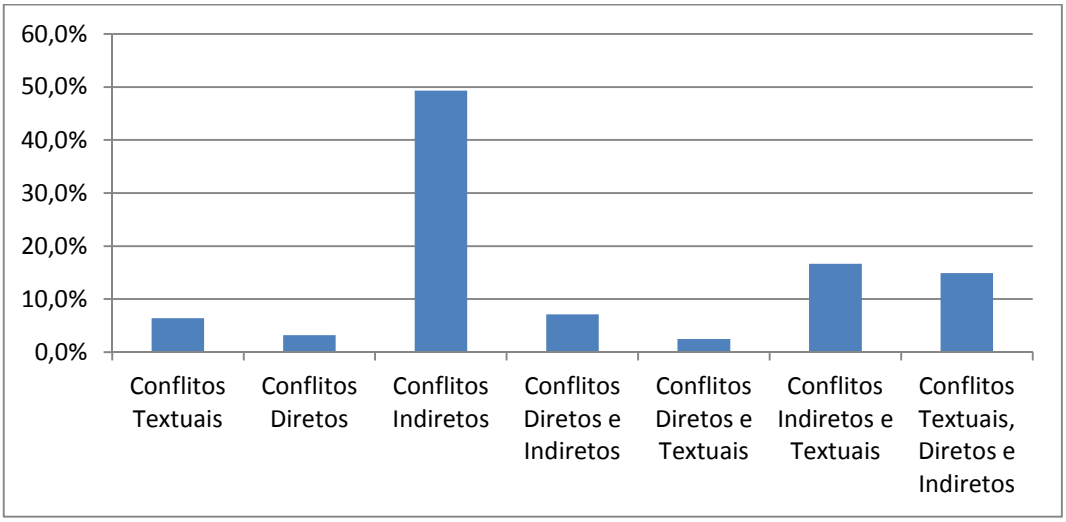


Figura 6-2 Percentual de Conflitos em Tarefas do *Target*

Resolução de Conflitos. Conforme dados apresentados nesta questão de pesquisa e sintetizados na Tabela 6-17 verifica-se que: (i) 71,3% das tarefas do *source* podem ser incorporadas ao *target* com segurança uma vez que a análise não detectou nenhum conflito nas mesmas; (ii) 8,9% das tarefas são resolvidas de forma automatizada; e (iii) 14,6% de forma semi-automatizada demandam a realização de testes para a sua condução. Por fim, apenas 5,2% de todas as tarefas do *source* devem ser resolvidas através da resolução manual por parte do usuário.

Tipo de Ocorrência	**Percentual**	**Tipo de Resolução**
Sem conflitos identificados	71,3%	Automatizada – Merge seguro
Somente conflitos textuais	8,9%	Automatizada
Conflitos indiretos e textuais	14,6%	Semi-automatizada
Com conflito direto	5,2%	Manual
Total	100%	

Tabela 6-17 Sumário da Resolução de Conflitos por Tarefa

Os sistemas de controle de versão tradicionais são capazes de executar *merges* e incorporar as mudanças nos artefatos que não possuem conflitos léxicos, ou seja, aqueles que não tiverem modificações concorrentes nos mesmos arquivos. Porém, a abordagem proposta nesta tese indica quais tarefas são seguras para conciliação. Os sistemas de controle de versão realizam apenas análise léxica nos artefatos, sem agrupar por tarefas, e exibindo-os visualmente destacando suas diferenças textuais. Após o conflito identificado, a única informação disponível é em qual artefato ele se manifestou, não há identificação da requisição de mudança específica (tarefa de desenvolvimento) que ele está relacionado, outras mudanças no código relacionadas a tal requisição, e que tipos de impactos em outras mudanças realizadas aquela evolução pode gerar. Desta forma, a utilização da abordagem sugerida para o processo de *merge* de LPS clonadas representa um avanço frente às metodologias tradicionais, pelo tratamento dado à evolução e às operações atômicas, e que representa 71,3% das tarefas do *source*.

As tarefas com conflitos textuais são resolvidas automaticamente pela abordagem, sem nenhuma intervenção do usuário. Elas representam 8,9% de todas as tarefas do *source*. As tarefas que possuem somente conflitos indiretos (8,5%) e as que possuem conflitos indiretos e textuais (6,1%) totalizam 14,6% das tarefa e são passíveis

de resolução semi-automatizada, ou seja, aquelas em que a abordagem resolve o conflito mas indica as tarefas dependentes para serem re-testadas pelo time de desenvolvimento. Já as tarefas que possuem algum tipo de conflito direto (2,2% somente diretos, 0,9% diretos e indiretos, 0,2% diretos e textuais e 1,9% todos os conflitos) totalizam 5,2% do total de tarefas que serão necessariamente objeto de resolução manual nas operações atômicas onde se manifestou o conflito direto.

6.5.1.3 QP-1.3 - Qual a frequência de conflitos que ocorrem por tarefa no source e no target?

A média de conflitos é uma variável importante a ser analisada, mas pode não representar a realidade para o caso em que poucas tarefas possuem muitos conflitos e a maioria das tarefas possuem poucos conflitos. As Tabela 6-18, Tabela 6-19 e a Figura 6-3 apresentam histogramas da frequência de ocorrência de conflitos nas tarefas. Mais da metade das tarefas do *source* possuem apenas um conflito. Já no *target*, mais conflitos se manifestam, com apenas 24,7% das tarefas com apenas um conflito e com 43,7% das tarefas possuindo cinco ou mais conflitos.

Total de Conflitos	**Percentual**	**Faixas**
1	56,6%	**56,6%**
2	13,5%	22,3%
3	5,4%	
4	3,4%	
5	2,8%	21,10%
10	7,5%	
20	4,6%	
Mais	6,2%	

Tabela 6-18 Frequência de ocorrência de conflitos - *Source*

Total de Conflitos	Percentual	Faixas
1	24,7%	24,7%
2	14,2%	31,6%
3	8,8%	
4	8,6%	
5	4,7%	**43,70%**
10	13,9%	
20	11,8%	
Mais	13,3%	

Tabela 6-19 Frequência de Conflitos – *Target*

A análise também foi realizada pelo tipo de conflito para investigar se a frequência ilustrada nas Tabelas 6-18 e 6-19 se mantém para cada tipo de conflito. A Figura 6-3 ilustra a distribuição por tipo. Observe que os percentuais de tarefas com apenas um conflito léxico (textual e direto) é 78% e 76%, respectivamente. Apenas 30% das tarefas possuem apenas um conflito indireto. Cerca de 40% das tarefas possuem 10 ou mais conflitos indiretos, conforme ilustrado na Figura 6-3 e Tabela 6-20. Conforme pode-se observar, a maioria das tarefas apresenta um número baixo de conflitos diretos e textuais, entretanto, no que diz respeito a conflitos indiretos, temos cerca de 40% das tarefas que apresentam 10, 20 ou mais conflitos.

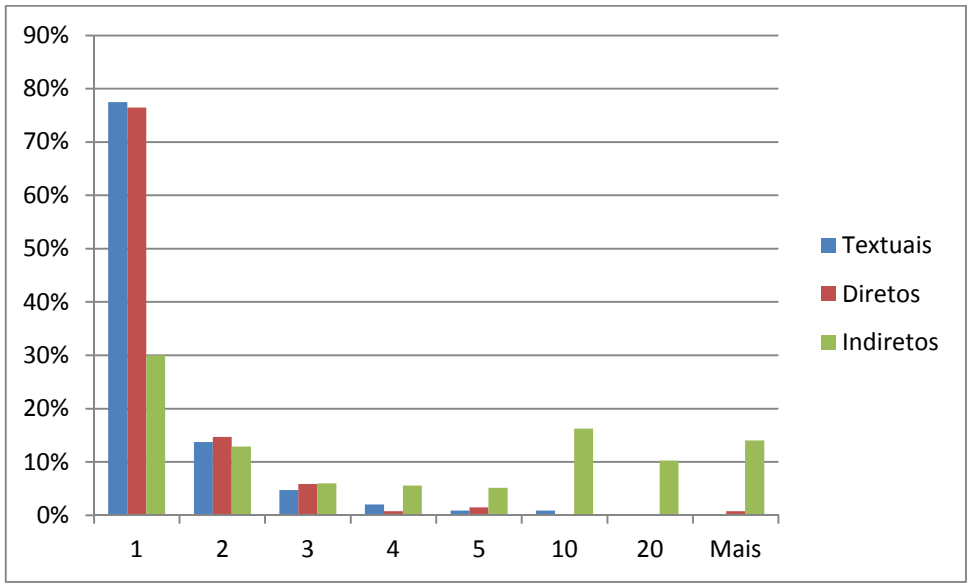


Figura 6-3 Distribuição de Conflitos no *Source*

Total de Conflitos	Textual	Direto	Indireto
1	78%	76%	30%
2	14%	15%	13%
3	5%	6%	6%
4	2%	1%	6%
5	1%	1%	5%
10	1%	0%	16%
20	0%	0%	10%
Mais	0%	1%	14%

Tabela 6-20 Distribuição de Conflitos no *Source*

A frequência do total de conflitos no *target* é ilustrado na Figura 6-4 e Tabela 6-21. Diferentemente do *source*, os percentuais de tarefas com apenas um conflito diminuem. Apenas os conflitos diretos possuem mais da metade das tarefas com apenas um conflito, os demais são apenas 21% textuais e 16% indireto. Os conflitos indiretos destacam-se também no *target* pelo fato de mais da metade das tarefas (51%) possuírem 10 ou mais conflitos, havendo assim muito conflitos identificados pela análise das dependências.

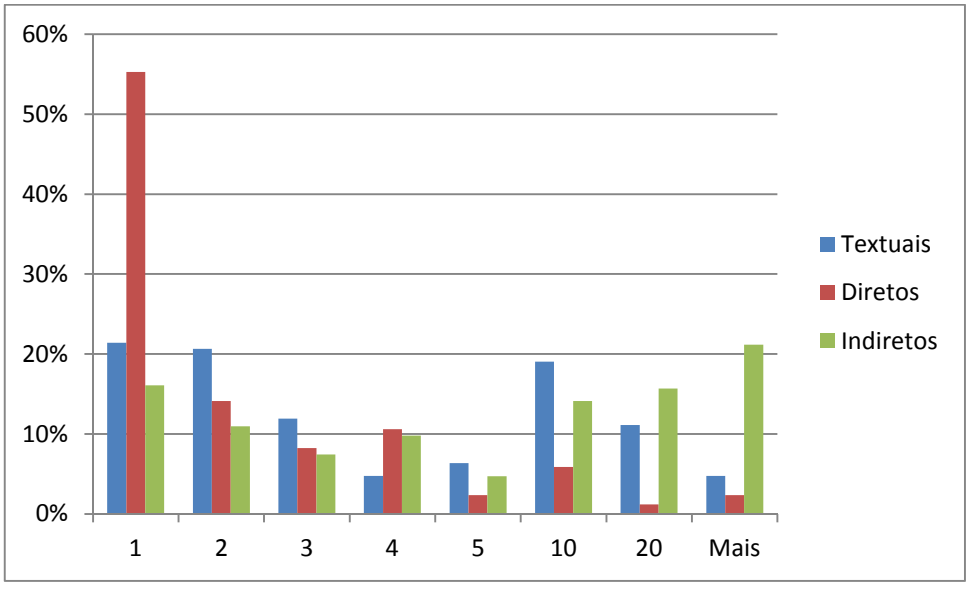


Figura 6-4 Distribuição de Conflitos no *Target*

Total de Conflitos	Textual	Direto	Indireto
1	21%	55%	16%
2	21%	14%	11%
3	12%	8%	7%
4	5%	11%	10%
5	6%	2%	5%
10	19%	6%	14%
20	11%	1%	16%
Mais	5%	2%	21%

Tabela 6-21 Distribuição de Conflitos no Target

6.5.1.4 QP-1.4 - Quantos e quais conflitos acontecem em cada operação atômica no source e no target

As tabelas Tabela 6-22 e Tabela 6-23 mostram o total de evoluções e o total de conflitos relacionados as operações atômicas de tais evoluções no *source* e no *target*. Considerando um total de 29.321 evoluções atômicas analisadas no *source*, 5,5% apresentaram algum tipo de conflito, sendo eles: 2% textuais, 0,6% diretos e 2,9% indiretos. Já no *target,* o mais importante a ser analisado, o percentual de conflitos encontrados em operações atômicas foi maior: em 33,7% das evoluções atômicas ocorreram conflitos, dos quais 5,7% textuais, 5,0% diretos e 23,1% indiretos.

	E1		E2		E3		E4		Soma
Total de Evoluções Atômicas	5924		2203		10597		10597		**29.321**
Sem Conflitos	5.632	95,1%	2.114	96,0%	10.268	96,9%	9.543	90,1%	**94,5%**
Conflitos Textuais	125	2,1%	31	1,4%	89	0,8%	371	3,5%	**2,0%**
Conflitos Diretos	64	1,1%	4	0,2%	14	0,1%	120	1,1%	**0,6%**
Conflitos Indiretos	103	1,7%	54	2,5%	226	2,1%	563	5,3%	**2,9%**

Tabela 6-22 Total de conflitos por operação atômica no *Source*

Target	E1		E2		E3		E4		Soma
Total de Evoluções Atômicas	354		429		429		2203		**3415**
Sem Conflitos	210	59,3%	361	84,1%	240	55,9%	1.447	65,7%	**66,3%**
Conflitos Textuais	30	8,5%	16	3,7%	24	5,6%	107	4,9%	**5,7%**
Conflitos Diretos	42	11,9%	4	0,9%	13	3,0%	89	4,0%	**5,0%**
Conflitos Indiretos	72	20,3%	48	11,2%	152	35,4%	560	25,4%	**23,1%**

Tabela 6-23 Total de conflitos por operações atômicas no Target

À primeira vista, o percentual de conflitos em operações atômicas (5,5% no *source* e 33,7% no *target*) pode parecer pouco significativo, porém em uma análise absoluta, são 1.614 operações no *source* e 1.151 no *target*. Ou seja, são milhares de operações atômicas para resolver conflitos e, a depender do tempo gasto para realizar cada uma, o tempo total resultante pode ser considerável e com alto custo para uma organização.

Resolução de Conflitos: A análise por operação atômica demonstrou que, em média, 94,5% das operações atômicas do *source* são identificadas como ausente de conflitos, 2% podem ser resolvidos de forma automatizada pela abordagem por representarem conflitos textuais, 2,9% são resolvidos de forma semi-automatizada (conflitos indiretos) e 0,6% são resolvidas manualmente pelo usuário (conflitos diretos).

6.5.1.5 QP-1.5 Qual a distribuição dos conflitos nas diversas operações atômicas?

As análises até então realizadas identificaram que tipos de conflitos ocorreram, o percentual de evoluções e tarefas conflitantes e com que frequência os conflitos ocorrem. Porém, é preciso investigar também como os conflitos se manifestam em quais operações atômicas. As Tabela 6-24 e Tabela 6-25 enumeram os conflitos por tipo de operação atômica. Observe que a maioria dos conflitos, 84,4% no *source* e 92,1% no *target* ocorre em atualizações de métodos. Em seguida a adição de métodos, com 4,8%, foi a operação que mais gerou conflitos no *target* e a atualização de atributos no *Source* com 5,0%.

Operação Atômica	**Sumário**				
Tipo de Conflito	**D**	**% Direto**	**I**	**% Indireto**	**% Geral**
Field Add	0	0,0%	28	2,7%	1,3%
Field Delete	2	0,9%	63	6,0%	3,4%
Field Update	4	1,7%	87	8,2%	5,0%
Method Add	3	1,3%	34	3,2%	2,3%
Method Delete	5	2,2%	53	5,0%	3,6%
Method Update	216	93,9%	790	74,9%	84,4%
Total	**230**	**100,0%**	**1055**	**100,0%**	**100,0%**

Tabela 6-24 Sumário de Conflitos de Operações Atômicas no *Source*

Operação Atômica	**Sumário**				
Tipo de Conflito	**D**	**% Direto**	**I**	**% Indireto**	**% Geral**
Field Add	0	0,0%	0	0,0%	0,0%
Field Delete	2	1,4%	0	0,0%	0,7%
Field Update	4	2,7%	2	0,7%	1,7%
Method Add	0	0,0%	28	9,6%	4,8%
Method Delete	2	1,4%	0	0,0%	0,7%
Method Update	140	94,6%	261	89,7%	92,1%
Total	**148**	**100,0%**	**291**	**100,0%**	**100,0%**

Tabela 6-25 Sumário de Conflitos de Operações Atômicas no Target

Os conflitos textuais não aparecem nas tabelas acima pois eles ocorrem apenas quando classes evoluem em operações atômica distintas. Desta forma, quando um conflito textual ocorre não há conflito direto entre as operações atômicas.

6.5.1.6 QP-1.6 - Qual a profundidade média no grafo de chamados na ocorrência de conflitos indiretos?

Os conflitos indiretos são computados através do percurso no grafo de chamadas dos métodos envolvidos com uma determinada tarefa. A partir da identificação de uma mudança verifica-se se ela faz parte do grafo de chamadas de alguma outra tarefa modificada, caso sim, identifica-se um conflito indireto. O estudo utilizou profundidade seis como nível máximo de busca de conflitos indiretos . A cada identificação de conflito a profundidade foi armazenada no modelo de conflitos. Desta forma, a partir da

análise das estatísticas, identificou-se que 58,1% de todos os conflitos acontecem na profundidade um, indicando que a maioria dos conflitos indiretos ocorrem no método ou atributo imediatamente consecutivo do grafo de chamada. Os dados obtidos são enumerados na Tabela 6-26.

Profundidade	***E1***	***E2***	***E3***	***E4***	***Total***	***Percentual***
1	118	90	621	2530	3359	**58,1%**
2	43	13	50	627	733	12,7%
3	11	5	57	584	657	11,4%
4	6	6	78	394	484	8,4%
5	0	3	42	267	312	5,4%
6	0	5	21	215	241	4,2%

Tabela 6-26 Profundidade do conflito indireto no grafo de chamadas

6.5.2 QP2 - Que tipos de tarefas – correção de *bugs*, melhoria de funcionalidades, novos casos de usos – trazem mais conflitos nas linhas de produto de software clonadas?

Esta seção explora os resultados da questão de pesquisa QP2 cujo foco é a análise de como os conflitos se manifestam nos tipos de tarefas de correção de bugs, novas funcionalidades e novos casos de usos.

6.5.2.1 QP-2.1 - Como os conflitos se manifestam nos diferentes tipos de tarefas no source e no target?

Os tipos de tarefas de desenvolvimento possuem tratamentos diferenciados no processo de engenharia de *software*. Tarefas de correção de *bugs* normalmente são executadas sob prazos mais curtos, uma vez que representam comportamentos incorretos em versões já liberadas ao usuário final. Tarefas de novos casos de usos e melhoria de funcionalidades existentes seguem o fluxo completo de engenharia (análise, projeto, implementação, teste, publicação). Uma questão de pesquisa no contexto dessa tese de doutorado é como a natureza da tarefa se relaciona com o total de conflitos ocorridos e quais os tipos de conflitos que mais ocorrem quando integrando linhas de produto de software clonadas.

As Tabelas Tabela 6-27 e Tabela 6-28 sintetizam o total de tarefas de desenvolvimento ocorridas no *source* e no *target* e os percentuais de conflitos para cada tipo de tarefa. Pela análise dos dados apresentados verifica-se um percentual maior de conflitos (33,3%) no *source* para as tarefas de melhoria de funcionalidades. Já no *target* as tarefas que mais manifestaram conflitos foram as de desenvolvimento de novos casos de usos (87,1%) e de melhoria de funcionalidades (81,8%). De forma geral, considerando o total de conflitos, verifica-se uma distribuição equilibrada entre os diferentes tipos de tarefas, mas é necessário uma análise mais detalhada do tipo de conflito que ocorreu para poder entender melhor os dados em questão.

Tipo de Tarefa	Total de Tarefas	Tarefas com Conflitos	%
Correção de Bugs	1338	340	25,4%
Melhoria de Funcionalidades	1142	380	33,3%
Novos Casos de Usos	102	23	22,5%
Totais	**2582**	**743**	**28,8%**

Tabela 6-27 Evoluções e Conflitos por Tipo de Tarefa - *Source*

Tipo de Tarefa	Tarefas	Tarefas com Conflitos	%
Correção de Bugs	298	191	64,1%
Melhoria de Funcionalidades	77	63	81,8%
Novos Casos de Usos	31	27	87,1%
Totais	**406**	**281**	**69,2%**

Tabela 6-28 Evoluções e Conflitos por Tipo de Tarefa - *Target*

As Tabelas Tabela 6-27 e Tabela 6-28 computam o total de tarefas com conflitos, mas não detalham que tipos de conflitos são esses e quantos são estes conflitos. Buscando entender melhor esse aspecto, a análise foi estendida para levar em consideração os tipos de conflitos (diretos, textuais e indiretos) das tarefas bem como a quantidade em que ocorrem.

Conflitos Diretos				
Tipo de Tarefa	*Source*		*Target*	
Correção de *bugs*	87	43,1%	83	41,1%
Melhoria de Funcionalidades	110	**54,5%**	102	**50,5%**
Novos Casos de Usos	5	2,5%	17	8,4%
	202	100,0%	**202**	100,0%

Tabela 6-29 Detalhamento dos Conflitos Diretos pelos tipos de tarefas

A Tabela 6-29 detalha a ocorrência dos conflitos diretos. É possível observar que os tipos de tarefas conflitam em percentuais similares no *source* e no *target,* ou seja, no que tange os conflitos diretos as tarefas, em geral, conflitaram em percentuais similares no *source* e no *target*.

Conflitos Textuais				
Tipo de Tarefa	*Source*		*Target*	
Correção de *bugs*	202	33,7%	309	**51,5%**
Melhoria de Funcionalidades	373	**62,2%**	221	36,8%
Novos Casos de Usos	25	4,2%	70	11,7%
	600	100,0%	**600**	100,0%

Tabela 6-30 Detalhamento dos Conflitos Textuais pelos Tipos de Tarefas

A Tabela 6-30 detalha os conflitos textuais por tipo de tarefa. As tarefas de melhoria de funcionalidades do *source* representam a maior ocorrência, com 62,2% dos conflitos. Já no *target* é a a correção de *bugs* com 51,5%. Ou seja, muitas tarefas de melhoria de funcionalidades do *source* conflitaram com tarefas de correção de *bugs* no *target*. Neste cenário, verifica-se que muitas funcionalidades que estão sendo evoluídas pelo *source* ainda contém *bugs* que estão sendo resolvidas no *target* (*clone*) em outros métodos e/ou atributos que não aqueles que estão sendo evoluídos. A publicação de melhoria de funcionalidades somente após a correção de todos os *bugs* nelas contidas poderia diminuir essa quantidade de conflitos textuais.

Conflitos Indiretos				
Tipo de Tarefa	*Source*		*Target*	
Resolução de *Bugs*	1657	31,3%	3288	**62,1%**
Melhoria de Funcionalidades	3467	**65,5%**	1293	24,4%
Novos Casos de Usos	170	3,2%	710	13,5%
	5294	100,0%	**5294**	100,0%

Tabela 6-31 Detalhamento dos Conflitos Indiretos pelos Tipos de Tarefas

A Tabela 6-31 detalha os conflitos indiretos nos três tipos de tarefas. De forma similar, ao que se verificou nos conflitos textuais, muitas das atualizações de funcionalidades do *source* levam a impactos indiretos nas resoluções de erros. Um total de 62,1% das tarefas de correção de *bugs* tiveram algum tipo de conflito indireto no *target*, gerados, em alguma parte, pelas atualizações de funcionalidades no *source*, já que este representa 65,5% dos conflitos. A diminuição deste fenômeno pode ser obtida com um maior investimento do *source* na estabilidade do software e a diminuição da necessidade de correção de *bugs* na linha de produto clonada.

Todos os conflitos				
Tipo de Tarefa	*Source*		*Target*	
Resolução de *Bugs*	1948	31,9%	3680	**60,2%**
Melhoria de Funcionalidades	3960	**64,8%**	1631	26,7%
Novos Casos de Uso	204	3,3%	801	13,1%
	6112	100,0%	**6112**	100,0%

Tabela 6-32 Todos os Conflitos por tipo de tarefas

Conforme ilustra a Tabela 6-32, os conflitos ocorrem, no geral, em melhoria de funcionalidades no *source* e em correção de bugs no *target*. Os conflitos de novos casos de uso, apesar de representar um pequeno percentual (3,3% *source* e 13,1% no *target*) ainda indica que melhores técnicas devem ser usadas no isolamento de artefatos. Tais conflitos, por se tratar de novas funcionalidades, deveriam ser bastante reduzidos e se manifestar apenas pelo fato de ambos os lados estarem reutilizando determinadas classes para desenvolver novas funcionalidades. Certamente não é possível ter um isolamento completo já que algumas estruturas, tais como, as classes de domínio, necessitam ser modificadas para novos casos de uso, mas a incidência de conflitos diretos nestes casos pode indicar práticas não adequadas de implementação no que diz respeito ao isolamento dos artefatos criados. A análise detalhada dos conflitos léxicos (textuais e diretos) indicou que a maioria ocorrem em estruturas compartilhadas, tais como, a classe `AcessoMenu` – utilizada para controlar autorizações na camada de apresentação em todos os módulos e a classe `SigaaSubSistemas` que contém a lista de constantes de cada módulo do sistema. Uma das tarefas do *source* adicionou um novo caso de uso de um questionário voltado para coordenadores de cursos e bolsistas. Este novo caso de uso manifestou um conflito direto, três textuais e sete indiretos. O conflito direto foi no método `TelasPosSelecaoVinculos.java$String#iniciar()`, um método chamado no entrada do sistema e que é modificado por muitas operações. Já os textuais foram classes `AcessoDiscente`, `DadosDiscente` e `SigaaListaComando`, estruturas da camada de segurança e normalmente modificada por novos casos de usos. No apêndice A estão localizadas figuras que detalham a distribuição de conflitos por tipo de tarefa no *source* do *target*.

As Tabelas Tabela 6-33 e Tabela 6-34 ilustram as médias de conflitos manifestados por cada operação atômica e tarefa do clone (*target*). Observe que a cada modificação atômica realizada no clone em média são manifestados 10,41 conflitos dos vários tipos, sendo sub-dividida pelos tipos de tarefas conforme ilustra a tabela. A cada tarefa de desenvolvimento no clone em média são manifestados 15,05 conflitos na reconciliação.

Tipo de Tarefa	Evoluções Atômicas no Clone (*Target*)	Total de Conflitos	Média de Conflitos por Evolução Atômica
Correção de *bugs*	420	3680	8,76
Melhoria de Funcionalidades	115	1631	14,18
Novos Casos de Uso	52	801	15,40
Total	**587**	**6112**	**10,41**

Tabela 6-33 Média de conflitos por tipo de tarefa e evolução e por evolução atômica

Tipo de Tarefa	Tarefas no Clone (*Target*)	Conflitos	Média de Conflitos por Tarefa
Correção de *Bugs*	298	3680	12,35
Melhoria de Funcionalidades	77	1631	21,18
Novos Casos de Uso	31	801	25,84
Total	**406**	**6112**	**15,05**

Tabela 6-34 Média de Conflitos por tipo de tarefa

6.5.2.2 QP-2.2 - Qual o percentual de manifestação das diferentes combinações de conflitos nas tarefas de correção de bugs, novas funcionalidades e aprimoramentos?

A análise da QP-2.1 identificou o total de conflitos por tipo de tarefa, seu percentual de ocorrência e suas médias, porém para analisarmos a resolução de conflitos por tarefa é preciso investigar como os vários tipos de conflitos se manifestam nas tarefas de resolução de *bugs*, melhoria de funcionalidades e novos casos de usos. Isso permite analisar também quais destes conflitos podem ser resolvidos automaticamente,

semi-automaticamente ou manualmente. As Tabelas Tabela 6-35 e Tabela 6-36 detalham as tarefas com conflitos indicando os tipos de combinação de conflitos que elas mantém. Por exemplo, no que se refere a tarefas de resolução de *bugs*, cerca de 75% apresentam conflitos textuais e/ou indireto. Quanto as tarefas de novos casos de usos, verifica-se que grande parte delas possuem conflitos textual e/ou indireto (82%). Finalmente, no que se refere as tarefas de melhorias de funcionalidades cerca de 85% representam tarefas com conflitos indiretos e/ou textuais.

	Correção de *Bugs*		**Melhoria de Funcionalidades**		**Novos Casos de Uso**	
Conflitos Textuais	109	**32,1%**	106	27,9%	12	**52,2%**
Conflitos Diretos	47	13,8%	9	2,4%	0	0,0%
Conflitos Indiretos	98	28,8%	121	**31,8%**	4	17,4%
Conflitos Diretos e Indiretos	14	4,1%	8	2,1%	0	0,0%
Conflitos Diretos e Textuais	10	2,9%	6	1,6%	2	8,7%
Conflitos Indiretos e Textuais	52	15,3%	94	24,7%	3	13,0%
Conflitos Textuais, Diretos e Indiretos	10	2,9%	36	9,5%	2	8,7%
Totais	340	100%	380	100%	23	100%

Tabela 6-35 Distribuição dos Conflitos por Tipo de Tarefa - *Source*

Target	**Correção de *Bugs***		**Melhoria de Funcionalidades**		**Novos Casos de Uso**	
Conflitos Textuais	13	6,8%	3	4,8%	2	7,4%
Conflitos Diretos	8	4,2%	1	1,6%	0	0,0%
Conflitos Indiretos	103	**53,9%**	3	4,8%	13	**48,1%**
Conflitos Diretos e Indiretos	9	4,7%	21	**33,3%**	3	11,1%
Conflitos Diretos e Textuais	3	1,6%	10	15,9%	0	0,0%
Conflitos Indiretos e Textuais	33	17,3%	8	12,7%	6	22,2%
Conflitos Textuais, Diretos e Indiretos	22	11,5%	17	27,0%	3	11,1%
Totais	191	100%	63	100%	27	100%

Tabela 6-36 Distribuição dos Conflitos por Tipo de Tarefa - *Target*

Resolução de Conflitos nas Tarefas. Para analisar a resolução dos conflitos nas tarefas deve-se verificar especificamente os conflitos ocorridos no *source*, pois ele representa a origem das tarefas a serem aplicadas no *target*. A motivação principal da QP2 é identificar de que forma e que tipos de conflitos as tarefas apresentam, assim permitindo a análise de quais destas tarefas podem ter resolução automatizada, semi-automatizada ou manual. A Tabela 6-37 enumera as formas de resolução por tipo de tarefa. Pode se observar que para um grande número de tarefas não são encontrados conflitos – entre

67% e 77% – dependendo do tipo da tarefa. Em torno de 10% de todas as tarefas de cada tipo podem ser resolvidas automaticamente, por apresentarem apenas conflitos textuais. Vale lembrar que usando ferramentas de merge baseada em análise léxica disponíveis em ferramentas de controle de versão, os conflitos de tais tarefas precisam ser resolvidos manualmente. As tarefas que podem ser resolvidas de forma semi-automatizada variam bastante – de cerca de 7% a 19% - dependendo do tipo da tarefa. Finalmente, o número de tarefas que tem que ser resolvidas manualmente, por apresentarem conflitos diretos, varia em torno de 4% a 7% dependendo do tipo de tarefa considerado. Isso mostra que a abordagem proposta na tese permite a resolução automática e semi-automática de mais de 93% das tarefas das LPS clonadas investigadas no nosso estudo.

Correção de *Bugs*		
Tipo de Ocorrência	**Percentual**	**Tipo de Resolução**
Sem conflitos identificados	70,1%	Automatizada – Merge seguro
Somente conflitos textuais	9,6%	Automatizada
Conflitos indiretos e textuais	13,2%	Semi-automatizada
Com conflito direto	7,1%	Manual
Melhoria de Funcionalidades		
Tipo de Ocorrência	**Percentual**	**Tipo de Resolução**
Sem conflitos identificados	66,7%	Automatizada – Merge seguro
Somente conflitos textuais	9,3%	Automatizada
Conflitos indiretos e textuais	18,8%	Semi-automatizada
Com conflito direto	5,2%	Manual
Novos Casos de Usos		
Tipo de Ocorrência	**Percentual**	**Tipo de Resolução**
Sem conflitos identificados	77,5%	Automatizada – Merge seguro
Somente conflitos textuais	11,8%	Automatizada
Conflitos indiretos e textuais	6,9%	Semi-automatizada
Com conflito direto	3,9%	Manual

Tabela 6-37 Resolução de Conflitos por Tipo de Tarefa

6.5.3 QP3 - Onde e como os conflitos ocorrem ao longo das camadas da LPS?

Esta seção explora os resultados da questão de pesquisa QP3 cujo foco é a análise dos conflitos em termos de camadas arquiteturais da LPS.

6.5.3.1 QP-3.1 - Em que camadas/módulos da LPS, os conflitos de código ocorrem na LPS, tanto no source quanto no target?

Em sistemas modernos desenvolvidos usando uma arquitetura em camadas (Fowler, 2012), as modificações devem seguir as regras de dependência estabelecida entre as camadas, fazendo com que o *software* evolua de forma robusta e controlável. Entender em que camadas ocorrem os conflitos é importante, pois as técnicas e padrões disponíveis buscam tratar problemas comuns e específicos de cada camada, tais como: web, negócio, entidades, acesso a dados, e outros.

O objetivo da QP3 é investigar em que camadas estes conflitos ocorrem, para que, a partir de tal informação, se possa entender onde e como tais conflitos ocorrem, e analisar se padrões específicos de implementação daquela camada podem ajudar a reduzir ou resolver conflitos. As Tabelas Tabela 6-38 e Tabela 6-39 enumeram os diferentes tipos de conflitos coletados no estudo no *source* e no *target*, respectivamente. Para cada conflito foi identificada a classe onde o conflito ocorreu, assim como a camada da qual tal classe faz parte, a partir dos padrões arquiteturais definidas para a LPS na nomenclatura dos pacotes. As classes `br.*.negocio.*` pertencem à camada de negócio, as `br.*.dao.*` ao pacote de acesso a dados, as `br.*.dominio.*` à camada de entidade (domínio), a `br.*.web.*`, `br.*.struts.*` e `br.*.jsf.*` pertencem à camada web, e os demais pacotes, tais como utilitários, *helpers*, segurança, dentre outros, foram catalogados como outras camadas.

Tipo de Conflito	Direto	Textual	Indireto	Média Geral
Camada de Negócio	15,3%	18,1%	11,2%	14,7%
Camada de Acesso a Dados	13,9%	30,7%	4,9%	16,3%
Camada de Entidade (Domínio)	5,9%	13,8%	**34,1%**	17,8%
Outras camadas	9,9%	5,8%	4,8%	6,8%
Camada Web	**55,0%**	31,6%	**45,1%**	**44,4%**

Total de Conflitos	100,0%	100,0%	100,0%	100,0%

Tabela 6-38 Conflitos por Camada - *Source*

A partir da análise da tabela 6-37 pode-se identificar que a camada Web foi a que apresentou mais conflitos (44,4%), seguidas pela camada de Entidade, Acesso a Dados e Negócio, que obtiveram percentuais próximos: 17,8%, 16,3% e 14,3%. As outra camadas tiverem 6,4% do conflitos. A camada Web é a líder em todos os tipos de conflitos, mas também é a camada mais modificada. A estrutura do framework JavaServer Faces utilizado nesta camada explica alguns dos conflitos manifestados e será discutido adiante neste capítulo. O alto número de conflitos obtido pela camada de Entidade foi motivada principalmente pelos conflitos indiretos, dos quais 34,1% são manifestados nesta camada. Vale ressaltar que pelo fato de muitos desses métodos que apresentam conflitos indiretos da camada de Entidade serem métodos de acesso, o impacto indireto de sua modificação não tende a ser muito crítico, principalmente quando não envolver mudanças de tipos.

A análise dos conflitos diretos demonstra que a maioria deles ocorreu na camada Web, com 55%, seguida das camadas de Negócio e Acesso a Dados, com 15,3% e 13,9%. O uso do padrão *EJB Command* (Crupi et al, 2004) contribui para a ocorrência de conflitos diretos na camada de negócio uma vez que todo processador de lógica de negócio tem como método principal o `execute(Movimento)`. A aplicação da refatoração de extração de código de tal método `execute()` de classes processador da camada de negócio para diferentes métodos com propósito bem definido pode contribuir para redução de tais conflitos diretos.

No que se refere a análise dos conflitos textuais a camada Web também apresentou um maior percentual de conflitos (31,6%) juntamente com a camada de Acesso a Dados (30,7%). Isto significa que a LPS clone (*target*) cria vários novos métodos de acesso a dados que pertencem as mesmas classes modificadas pelo *source* (as classes *Data Access Object*), mas que são independentes e não conflitam entre si. Estas evoluções ocorrem pela necessidade de adaptação dos critérios de consulta na base de dados ou a criação de novas consultas devido a mudança de requisitos para um novo produto clonado. Neste cenário, a utilização do padrão de projeto *Abstract Factory* (Gamma et al, 1994), ao invés do *Factory Method* (Gamma et al, 1994) para implementação das classes DAO (*Data Access Object*) (Malks et al, 2002), poderia

diminuir estes conflitos, mas, em compensação, dificultaria a reconciliação, já que cada clone teria sua implementação própria dos DAO. Outra alternativa é herdar as classes DAO do *source* em novas classes no clone e adicionar os novos métodos na classe filha, diminuindo assim os conflitos textuais. Vale lembrar que estas recomendações são sobretudo para ambientes de desenvolvimento que utilizam ferramentas de análise léxica para realização de merge de código, tais como, as disponíveis em sistemas de gerenciamento de versões atuais. Isso porque no caso da abordagem proposta nesta tese, a resolução de conflitos textual é automática.

A Tabela 6-39 enumera os conflitos ocorridos em camadas no *target*. Os dados para os conflitos direto e textual são exatamente iguais aos da Tabela 6-38, que enumera o *source*. Isto ocorre pois nos conflitos direto e textual sempre o conflito ocorre entre o mesmo artefato no *source* e no *target*, o que naturalmente representa a mesma camada. A diferença ocorre nos conflitos indiretos, porém com percentuais bem similares, pois, a abordagem avalia conflitos indiretos tanto no caminho *source-target* como no *target-source*.

Tipo de Conflito	Direto	Textual	Indireto	Média Geral
Camada de Negócio	15,3%	18,1%	11,8%	15,1%
Camada de Acesso a Dados	13,9%	**30,7%**	4,5%	16,4%
Camada de Entidade (Domínio)	5,9%	13,8%	34,9%	18,2%
Outras camadas	9,9%	5,8%	2,3%	6,0%
Camada Web	55,0%	31,6%	46,5%	**44,3%**
Total de Conflitos	100,0%	100,0%	100,0%	100,0%

Tabela 6-39 Sumário de Conflitos por Camada - *Target*

Os conflitos na camada Web são mudanças realizadas nas classes que representam *Managed Beans*, ou seja, elementos da subcamada de controle do padrão *Modelo-Visão-Controle* (Buschmann et al, 1996), responsáveis pelo tratamento das requisições submetidas através de formulários Web. Tais classes são normalmente impactadas em decorrência da mudança de visualização. Isto significa que as LPS clonadas alteram significativamente as visualizações (páginas JSP) e como decorrência destas evoluções a camada Web é impactada. Os conflitos na camada de domínio – 17,8% – indicam que o domínio da LPS não é genérico o suficiente, para permitir o armazenamento de todas as informações necessárias. Sendo assim, os clones necessitam

adicionar, modificar ou retirar campos para adequar ao seu modelo de negócio. As localizadas na subseção desta questão de pesquisa no apêndice A, ilustram graficamente a distribuição de conflitos por camada.

Com a alta incidência de conflitos diretos na camada Web é possível diminuir estes conflitos com algumas recomendações no desenvolvimento:

- Diminuir o tamanho dos métodos de controle das classes *Managed Beans*, criando novos métodos para divisão do fluxo do controle. Neste caso, como a funcionalidade do controle seria realizada por mais métodos, ao invés de um grande método com alta complexidade de fluxo, espera-se diminuir a ocorrência de conflitos diretos.
- Utilizar o padrão *Template Method* (Gamma et al, 1994) nos *Managed Beans* mais conflitantes, identificando possíveis variabilidades nos algoritmos de controle. Desta forma, criando *templates* de algoritmos de controle com implementação em subclasses, possibilitaria aos clones a criação de novas classes com a implementação do comportamento específico ao invés de adaptar de forma invasiva o comportamento das classes atuais.
- Evitar o uso de *Managed Beans* extensos, dividindo sua responsabilidade em mais classes de controle. É possível encontrar no código fonte das LPS analisadas, classes de controle com uma grande quantidade de linhas de código, e que são responsáveis pelo controle de vários passos ou formulários da interface gráfica. Uma recomendação seria dividir sua responsabilidade, criando *Managed Beans* auxiliares por passos ou formulários específicos.

A camada de domínio/entidades manifestou uma baixa incidência de conflitos diretos, pois em geral, as adaptações são para adicionar novas informações ao modelo. Apenas 5,9% dos conflitos diretos ocorreram nesta camada e, após análise, constatou-se que a maioria ocorre devido à necessidade de instanciação de campos que possuem vinculação direta com a interface gráfica, o *binding* do JSF.

As Tabelas Tabela 6-40 e Tabela 6-41 enumeram o total de evoluções atômicas ocorridas nas camadas em cada uma das evoluções clonadas. A maioria das evoluções ocorrem na camada Web, com 50,6% de todas as evoluções atômicas, seguida pelas camadas de Entidade, Acesso a Dados, Negócio e Outras. Pode-se observar que a

ordem das camadas que mais deram conflitos do *source* e *target* (Web, Entidade, Acesso a Dados, Negócio e Outros) coincide com a ordem das camadas que mais sofreram evoluções. Se analisarmos a sequência de evolução das linhas de produto clonadas (*target*), mesmo sendo desenvolvidas para instituições diferentes e por equipes diferentes, possuem a mesma sequencia de quantidade de modificações de operações atômicas por camadas (Web, Entidade, Acesso a Dados, Negócio e Outras).

Camada	**E1**		**E2**		**E3 e E4**		**Total**	
Web	2543	42,9%	1266	57,5%	5439	51,3%	9248	**50,6%**
Entidade	1133	19,1%	438	19,9%	2066	19,5%	3637	19,5%
Acesso a Dados	594	10,0%	284	12,9%	1430	13,5%	2308	12,1%
Negócio	677	11,4%	141	6,4%	1214	11,5%	2032	9,8%
Outras	977	16,5%	74	3,4%	448	4,2%	1499	8,0%
Total de Evoluções	5924	100%	2203	100,0%	10597	100,0%	18724	100,0%

Tabela 6-40 Evoluções por Camada – *Source*

Camada	**E1**		**E2 e E3**		**E4**		**Total**	
Web	147	41,5%	249	58,0%	1266	57,5%	1662	**52,3%**
Entidade	62	17,5%	83	19,3%	438	19,9%	583	18,9%
Acesso a Dados	55	15,5%	41	9,6%	284	12,9%	380	12,7%
Negócio	53	15,0%	53	12,4%	141	6,4%	247	11,2%
Outras	37	10,5%	3	0,7%	74	3,4%	114	4,8%
Total de Evoluções	354	100%	429	100%	2203	100,0%	2986	100,0%

Tabela 6-41 Evoluções por Camada - *Target*

As figuras localizadas na subseção do apêndice A desta questão de pesquisa ilustram a distribuição de conflitos no *target*.

6.5.3.2 QP-3.2 - Como cada tipo de tarefa do source (correção de bugs, melhoria de funcionalidades e novos casos de usos) conflita ao longo das camadas?

A análise por conflito realizada na QP-3.1 avalia onde os conflitos se localizam, porém sem agrupar por tarefa. O objetivo da QP-3.2 é analisar em quantas camadas uma dada tarefa conflita indicando que tipos de conflitos são esses. A Tabela 6-42

mostra que 39% das tarefas possuem conflitos em apenas uma camada, 18% em duas camadas, 14% em três camadas, 20% em quatro camadas e 9% em todas as camadas. A Tabela 6-43 analisa os dados no *target*, onde 48% conflitam em apenas uma camada, 20% em duas camadas, 15% em três camadas, 14% em quatro camadas e 3% em todas a camadas.

Combinação de Camadas	**Tarefas com Conflito Direto**		**Tarefas com Conflitos Textuais**		**Tarefas com Conflitos Indiretos**		**Média**
Conflitos em Apenas Uma Camada	66	49%	164	42%	161	34%	**39%**
Conflitos em Duas Camadas	26	19%	66	17%	92	19%	18%
Conflitos em Três Camadas	12	9%	47	12%	77	16%	14%
Conflitos em Quatro Camadas	20	15%	78	20%	103	22%	**20%**
Conflitos em Todas as Camadas	12	9%	34	9%	42	9%	9%
Total com Conflitos	**136**	100%	**389**	100%	**475**	100%	100%

Tabela 6-42 Distribuição dos Conflitos por camadas - *Source*

Combinação de Camadas	**Tarefas com Conflito Direto**		**Tarefas com Conflitos Textuais**		**Tarefas com Conflitos Indiretos**		**Média**
Conflitos em Apenas Uma Camada	36	46%	55	56%	124	46%	**48%**
Conflitos em Duas Camadas	20	26%	11	11%	57	21%	20%
Conflitos em Três Camadas	11	14%	16	16%	40	15%	15%
Conflitos em Quatro Camadas	7	9%	14	14%	40	15%	14%
Conflitos em Todas as Camadas	4	5%	3	3%	6	2%	3%
Total com Conflitos	**78**	100%	**99**	100%	**267**	100%	100%

Tabela 6-43 Distribuição de Conflitos por Camada - *Target*

No que tange a resolução manual de conflitos, ou seja, aqueles que são tarefas com conflitos diretos e não passíveis de tratamento pela abordagem verifica-se que cerca de 50% desses conflitos de resolução manual ocorrem somente em uma camada, facilitando assim a resolução por parte do desenvolvedor.

A Tabela 6-44 enumera a ocorrência de conflitos ao longo das camadas nas tarefas de correção de *bugs*. Verifica-se que 65% das tarefas conflitam em apenas uma camada, 16% em duas, 10% em três, 8% em quatro e 1% em todas. Isto mostra que, em

geral, uma tarefa de correção de *bugs* tende a modificar classes de uma mesma camada por se tratar de ajustes localizados para corrigir um determinado *bug*.

Correção de *Bugs* - *Source*							
Tipo de Conflito	**Diretos**		**Indiretos**		**Textuais**		**Média**
Conflitos em Apenas Uma Camada	50	68%	107	58%	116	72%	**65%**
Conflitos em Duas Camadas	15	21%	34	18%	20	11%	16%
Conflitos em Três Camadas	4	5%	22	12%	14	8%	10%
Conflitos em Quatro Camadas	3	4%	20	11%	10	5%	8%
Conflitos em Todas as Camadas	1	1%	2	1%	2	1%	1%
Total com Conflitos	73	100%	185	100%	162	88%	100%

Tabela 6-44 Distribuição de Conflitos em Camadas nas Tarefas de Correção de *Bugs*

A Tabela 6-45 enumera a ocorrência de conflitos ao longo das camadas considerando tarefas de melhorias de funcionalidades. Verifica-se que para este tipo de tarefas, existem uma boa distribuição em termos da quantidade de conflitos que ocorrem em cada camada. Vale ressaltar que 29% das tarefas apresentam conflitos em menos camadas (Web, Entidade, Acesso a Dados e Negócio), mostrando que tais tarefas demonstram ser mais complexas ao menos no que diz respeito a quantidade de conhecimento de código de várias camadas para sua resolução.

Melhoria de Funcionalidades - *Source*							
Tipo de Conflito	**Diretos**		**Indiretos**		**Textuais**		**Média**
Conflitos em Apenas Uma Camada	16	26%	70	20%	47	22%	21%
Conflitos em Duas Camadas	11	18%	68	19%	43	20%	19%
Conflitos em Três Camadas	8	13%	68	19%	28	13%	17%
Conflitos em Quatro Camadas	16	26%	104	30%	64	30%	**29%**
Conflitos em Todas as Camadas	11	18%	41	12%	32	15%	13%
Total com Conflitos	62	100%	351	100%	214	100%	100%

Tabela 6-45 Distribuição de Conflitos em Camadas nas Tarefas de Melhoria de Funcionalidades

A Tabela 6-27 ilustra que 22,5% das tarefas de novo casos de uso do *source* possuem conflitos para serem reconciliados no *target*. Neste caso, quando estes conflitos ocorrem, normalmente se manifestam em mais de uma camada. Pode-se observar também que 84% das tarefas conflitam em três ou mais camadas, entretanto boa parte dessas tarefas apresentam conflitos indiretos e textuais os quais podem ser resolvidos de forma automática. A Tabela 6-46 enumera a distribuição de conflitos por tarefas em novos casos de uso.

Novos Casos de Usos - *Source*							
Tipo de Conflito	**Diretos**		**Indiretos**		**Textuais**		
Conflitos em Apenas Uma Camada	0	0%	0	0%	1	5%	3%
Conflitos em Duas Camadas	0	0%	1	10%	3	16%	12%
Conflitos em Três Camadas	0	0%	2	20%	5	26%	21%
Conflitos em Quatro Camadas	1	25%	3	30%	4	21%	24%
Conflitos em Todas as Camadas	3	75%	4	40%	6	32%	**39%**
Total com Conflitos	4	100%	10	100%	19	100%	100%

Tabela 6-46 Distribuição de Conflitos em Camadas nas Tarefas de Novos Casos de Usos - *Source*

As Tabelas Tabela 6-47, Tabela 6-48 e Tabela 6-49 ilustram a distribuição dos conflitos em camadas nas tarefas de correção de *bugs*, melhoria de Funcionalidades e Novos Casos de Usos, respectivamente. Pode-se observar que os resultados do clone, apesar de percentualmente divergentes, os resultados são similares: as tarefas de correção de *bugs* conflitam em geral em uma camada, com 66% de ocorrência, as tarefas de melhoria de funcionalidades conflitam de forma distribuída entre as várias camadas, e as tarefas de novos casos de usos 58% delas conflitam em 3 ou mais camadas.

Correção de *Bugs* - *Target*							
Camada	**Diretos**		**Indiretos**		**Textuais**		
Conflitos em Apenas Uma Camada	27	**63%**	110	**64%**	45	73%	**66%**
Conflitos em Duas Camadas	11	26%	39	23%	8	13%	21%
Conflitos em Três Camadas	4	9%	16	9%	6	10%	9%
Conflitos em Quatro Camadas	0	0%	5	3%	1	2%	2%
Conflitos em Todas as Camadas	1	2%	2	1%	2	3%	2%
Total com Conflitos	43	100%	172	100%	62	100%	100%

Tabela 6-47 Distribuição de Conflitos em Camadas nas Tarefas de Correção de *bugs* - Target

Melhoria de Funcionalidades - *Target*							
Tipo de Conflito	**Diretos**		**Indiretos**		**Textuais**		**Média**
Conflitos em Apenas Uma Camada	8	26%	11	17%	8	27%	21%
Conflitos em Duas Camadas	9	**29%**	15	23%	2	7%	21%
Conflitos em Três Camadas	6	19%	14	22%	6	20%	21%
Conflitos em Quatro Camadas	4	13%	19	**29%**	9	**30%**	25%
Conflitos em Todas as Camadas	4	13%	6	9%	5	17%	12%
Total com Conflitos	31	100%	65	100%	30	100%	100%

Tabela 6-48 Distribuição de Conflitos em Camadas nas Tarefas de Melhoria de Funcionalidades - *Target*

Novos Casos de usos - *Target*							
Tipo de Conflito	**Diretos**		**Indiretos**		**Textuais**		**Média**
Conflitos em Apenas Uma Camada	1	17%	3	9%	2	18%	12%
Conflitos em Duas Camadas	0	0%	3	9%	1	9%	8%
Conflitos em Três Camadas	1	17%	10	29%	4	**36%**	29%
Conflitos em Quatro Camadas	3	**50%**	16	**47%**	4	**36%**	45%
Conflitos em Todas as Camadas	1	17%	2	6%	0	0%	6%
Total com Conflitos	6	100%	34	100%	11	100%	100%

Tabela 6-49 Distribuição de Conflitos em Camadas nas Tarefas de Novos Casos de Usos - Target

6.5.4 QP4 – Onde e como os conflitos ocorrem ao longo do core/variabilidades da LPS?

Esta seção explora os resultados da questão de pesquisa QP4 cujo foco é a análise de como os conflitos ocorrem considerando o núcleo da LPS e as variabilidades.

6.5.4.1 QP-4.1 – Em que parte da LPS (núcleo/variabilidades) ocorrem as evoluções e os conflitos em operações atômicas?

A evolução de uma LPS pode ocorrer tanto no núcleo como nas variabilidades. A adoção de técnicas de engenharia de LPS objetiva, principalmente, uma melhor gerência das variabilidades para permitir uma melhor acomodação de mudanças relacionadas ao contexto de negócio. Assim, no contexto do nosso estudo empírico, é importante identificar onde as evoluções estão ocorrendo, se no núcleo ou nas variabilidades da LPS. A Tabela 6-50 enumera as evoluções ocorridas em cada uma das diferentes evoluções de LPS clonadas. Observe que apenas 0,4% das evoluções, 5,9% das tarefas e 2,2% dos conflitos ocorreram nas variabilidades.

	E1		**E2**		**E3**		**E4**		**Média**
Evolução atômica									
Evoluções Atômicas – Núcleo	5875	99,2%	2197	99,7%	10556	99,6%	10556	99,6%	99,5%
Evoluções Atômicas-Variabilidade	49	0,8%	6	0,3%	31	0,3%	31	0,3%	**0,4%**
Tarefas									
Tarefas de	596	82,1%	233	97,9%	795	98,3%	795	98,3%	94,1%

Desenvolvimento – Núcleo									
Tarefa de Desenvolvimento – Variabilidade	130	17,9%	5	2,1%	14	1,7%	14	1,7%	**5,9%**
Conflitos									
Total de evoluções atômicas com Conflitos no Núcleo	342	97,2%	44	97,8%	137	98,6%	136	97,8%	97,8%
Total de evoluções atômicas com conflitos nas Variabilidades	10	2,8%	1	2,2%	2	1,4%	3	2,2%	**2,2%**

Tabela 6-50 Evoluções Atômicas, Tarefas e Conflitos no Núcleo e Variabilidades das LPS

As variabilidades que foram consideradas na análise da tabela 6-49 referem-se as vinte e quatro variabilidades explicitamente estão implementadas na LPS e catalogadas durante o desenvolvimento deste trabalho (Capítulo 3). No entanto, considerando o porte da LPS SIGAA e a quantidade de mudanças que são realizadas pelas várias instituições que a utiliza conclui-se que muitas características que estão sendo consideradas similaridades representam variabilidades na prática. Desta forma, para obter uma melhor análise do comportamento das evoluções e conflitos na LPS foi realizada uma análise qualitativa em cada uma das tarefas da evolução E4. Durante essa análise, cada tarefa e respectivos artefatos de código modificados foram analisados para verificar se tratava-se ou não de uma variabilidade. Após esta análise, a mineração de E4 foi re-executada e então se obteve os seguintes resultados: das 420 tarefas de melhoria de funcionalidades e novos casos de usos, 61 foram identificadas como variabilidades, representando 14% do total. A Tabela 6-51 enumera os resultados obtidos.

Evoluções atômicas		
Evoluções Atômicas – Núcleo	9.039	85,3%
Evoluções Atômicas- Variabilidade	1.553	**14,7%**
Tarefas		
Tarefas de Desenvolvimento – Núcleo	359	85,5%
Tarefa de Desenvolvimento – Variabilidade	61	**14,5%**
Conflitos		
Total de evoluções atômicas com Conflitos no Núcleo	563	69%

Total de evoluções atômicas com conflitos nas Variabilidades	253	**31%**

Tabela 6-51 Evoluções atômicas, Tarefas e Conflitos em Variabilidades

A análise qualitativa das características revelou que o cenário real de variabilidades é bem superior ao explicitamente implementado e catalogado. Considerando que a LPS SIGAA possui aproximadamente 1.850 funcionalidades o número de 24 variabilidades catalogadas representa um pouco mais de 1%. Dessa forma, é importante que o mantenedor da LPS *source,* no caso a UFRN, promova um investimento na criação explícita de variabilidades através do uso de técnicas composicionais e/ou anotativas, que permitam identificar, melhor gerenciar e, em alguns casos, diminuir a quantidade de conflitos entre as LPS.

6.5.4.2 QP-4.2 - Como os conflitos ocorrem no núcleo ou nas variabilidades da LPS

A Tabela 6-52 que considera as variabilidades catalogadas demonstra que 99,4% dos conflitos ocorrem no núcleo da LPS e apenas 0,6% nas variabilidades. A maioria dos conflitos em variabilidades ocorreram de forma indireta, isto significa que mudanças no núcleo estão impactando as variabilidades ou vice-versa. Por exemplo, na evolução clonada E4, os 18 conflitos indiretos manifestados representam mudanças nas variabilidades da LPS *source* que geraram conflitos indiretos com mudanças realizadas no núcleo do *target*. Ou seja, a abordagem identificou classes do núcleo do *target* que usam classes da LPS *source* com a variabilidade modificada, caracterizando assim conflitos indiretos.

	E1		**E2**		**E3**		**E4**		**Resumo**	
	Núcleo	*Var*	*Núcleo*	*Var*	*Núcleo*	*Var*	*Núcleo*	*Var*	*Núcleo*	*Var*
Conflitos Diretos	64	0	4	0	14	0	117	3	98,5%	2,2%
Conflitos Indiretos	171	7	121	1	862	7	4108	18	99,4%	0,6%
Conflitos Textuais	125	0	30	0	89	0	370	1	99,8%	0,2%
Totais	**360**	**7**	**155**	**1**	**965**	**7**	**4595**	**22**	**99,4%**	**0,6%**

Tabela 6-52 Distribuição dos Conflitos entre Core e Variabilidade da LPS

Se considerarmos a análise qualitativa realizada nas tarefas de melhoria de funcionalidades e novos casos de usos da evolução clonada E4 para categorizar se elas atuam sobre similaridades e variabilidades verifica-se que o total de conflitos reais em variabilidade é bem superior se comparados aos obtidos somente com as catalogadas. A Tabela 6-53 enumera que 26,35% dos conflitos são em variabilidades, sendo 92% (1114 no total) de conflitos indiretos. A análise das tarefas de correção de *bugs*, enumerada na Tabela 6-54, indica que 12,7% dos conflitos dos conflitos ocorrem em variabilidades, ou seja, os conflitos em variabilidades estão acontecendo mais frequentemente nas melhorias da LPS.

	E4		**E4 Percentual**	
	Núcleo	*Var*	*Núcleo*	*Var*
Conflitos Diretos	92	28	76,7%	23,3%
Conflitos Textuais	302	69	81,4%	18,6%
Conflitos Indiretos	3012	1114	73,0%	27,0%
Totais	**3406**	**1211**	**73,65%**	**26,35%**

Tabela 6-53 Conflitos em Variabilidades

	E4 *Bugs*		**E4 Percentual *Bugs***	
	Núcleo	*Var*	*Núcleo*	*Var*
Conflitos Diretos	30	3	91%	9%
Conflitos Textuais	76	32	70%	30%
Conflitos Indiretos	1078	137	89%	11%
Totais	**1187**	**172**	**87,3%**	**12,7%**

Tabela 6-54 Conflitos em Variabilidades em Correção de Bugs

6.5.4.3 QP-4.3 - No que diz respeito aos conflitos que ocorreram em variabilidades, onde foi maior a ocorrência em técnicas composicionais (padrões) ou anotativas (execução condicional)?

As variabilidades implementadas de forma explícita na LPS SIGAA e SIPAC utilizam as técnicas de execução condicional (anotativas) quanto padrões de projetos (composicionais). A partir do catálogo de variabilidades definido para tais LPS, este

estudo também identificou se os conflitos que ocorreram nas variabilidades foram implementados usando padrões de projeto ou execução condicional. A Tabela 6-55 mostra que a maior parte dos conflitos (91,9%) ocorreram sobre variabilidades implementadas usando execução condicional, com a ressalva que todos os conflitos diretos aconteceram através do uso desta técnica. A técnica de padrões manifestou um conflito textual e dois indiretos, o que mostrou que para os cenários do nosso estudo que todos eles poderiam ser resolvidos de forma automática através do uso da nossa abordagem. A análise detalhada dos conflitos nas variabilidades indicou que a maioria conflitos em técnicas de execução condicional ocorreram na camada Web e os que ocorreram em padrões foram manifestados na camada de negócio.

Tipo de Técnica	Total	Percentual
Execução Condicional	34	91,9%
Padrões	3	8,1%
Total	37	100,0%

Tabela 6-55 Conflitos nas Variabilidades por Tipo de Técnica

Para esta subquestão não é possível realizar a análise qualitativa das variabilidades da evolução clonada E4, pois trata-se dos conflitos ocorridos nas variabilidades catalogadas e explicitamente implementadas. O que se pode observar a partir da análise qualitativa é que as atuais técnicas podem ter sua utilização ampliada para outras funcionalidades da LPS. A partir da análise das 61 tarefas com variabilidades implícitas pode-se indicar que elas poderiam utilizar três situações de implementação:

i. Uso da técnica de execução condicional para implementação de características opcionais que atualmente estão sendo tratadas como mandatórias. Foram identificadas 33 tarefas nesta situação, 54% das variabilidades implícitas;
ii. Uso da técnica composicional de padrões de projetos que permitam implementar comportamentos diferentes em algoritmos e regras de negócios. Foram identificadas 17 tarefas nesta situação, 27% das variabilidades implícitas.
iii. Implementação combinada das técnicas anotativas e composicionais para implementação de características opcionais e que permitam variações de

granularidade grossa em sua implementação. Foram identificadas 11 tarefas nesta situação, 19% das variabilidades implícitas.

Para ilustrar um exemplo de variabilidade implícita considere a funcionalidade de Matrícula Extraordinária da LPS SIGAA. Esta funcionalidade é uma solicitação de matrícula em turmas remanescentes ocorrida em tempo real, após as etapas de processamento. Este comportamento é tratado na LPS como uma similaridade, porém trata-se de um comportamento da UFRN e diferente em outras universidades. A Figura 6-5 detalha a sua implementação, observe que todas as classes envolvidas de cada camada são concretas: (i) `MatriculaExtraOrdinariaMBean` – camada Web, (ii) `ProcessadorMatriculaExtraOrdinaria` – camada de negócio; e (iii) `MatriculaExtraOrdinariaDao` – camada de acesso a dados. Esta implementação poderia ser utilizada se efetivamente estivesse implementando uma similaridade entre os vários usuários da LPS, porém esta funcionalidade possui comportamentos diferentes em cada uma das diferentes instituições clientes que utilizam o SIGAA.

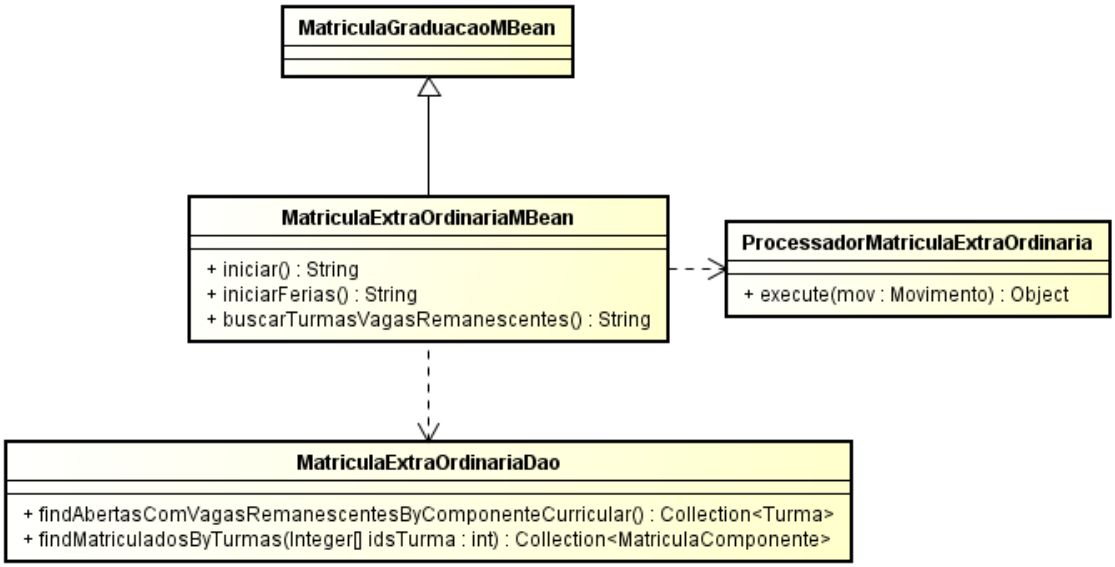


Figura 6-5 Variabilidade implícita de Matrícula Extraordinária

Por se tratar de uma variabilidade esta operação deveria utilizar as técnicas corretas de implementação. A Figura 6-6 ilustra uma recomendação de implementação

para esta funcionalidade, utilizando técnicas composicionais e anotativas. Os comportamentos da operação foram abstraídos em interfaces (`IMatriculaExtraOrdinariaController`, `IMatriculaExtraOrdinariaDao` e `IMatriculaExtraOrdinariaBusiness`) e as implementações passaram a ser uma variante de uma instanciação da LPS (produto) específica para a UFRN. A classe `MatriculaExtraOrdinariaFactory` se responsabiliza por obter as implementações corretas de acordo com parâmetros configurados na aplicação, conforme descrito no capítulo 3. O método `boolean isProductEnabled()` da classe `MatriculaExtraOrdinariaFactory` identifica se a funcionalidade de matricula extraordinária está habilitada para um determinado produto, implementando uma característica opcional através da técnica de execução condicional. As LPS estudadas nesta tese possuem diversos exemplos como este que poderiam ser transformados em variabilidades explicitamente implementadas.

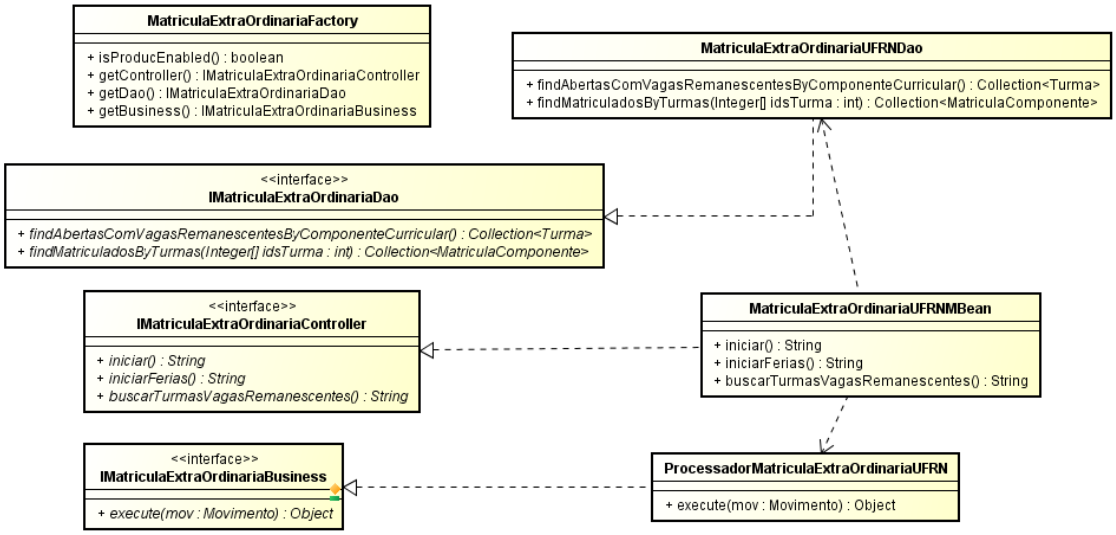


Figura 6-6 Implementação sugerida da variabilidade

Como trabalho futuro complementar à pesquisa desta tese de doutorado, pretende-se refatorar as variabilidades acima mencionadas usando as técnicas recomendadas e, em seguida, avaliar de que forma os conflitos acontecem. O objetivo principal é buscar evidências que o uso da técnica anotativa de execução condicional pode levar a geração de mais conflitos (sobretudo diretos) do que a técnica composicional de padrões de projeto.

6.5 Limitações e Ameaças à Validade

O estudo empírico abordado neste capítulo explorou a análise dos tipos de conflitos, como os tipos de tarefas se relacionam com os conflitos, onde eles ocorrem em termos de camadas arquiteturais e como se manifestam ao longo do núcleo e variabilidades da LPS. No entanto, cabe destacar algumas limitações: (i) o estudo precisa também ser replicado para outras LPS clonadas de sistemas de informação web, de forma a buscar ampliar e generalizar os resultados para LPS deste domínio (ou outros de interesse); (ii) o estudo considerou um número reduzido de variabilidades explícitas implementadas nas LPS, o que limitou a análise da relação entre os conflitos ocorridos e o núcleo/variabilidades das LPS; (iii) não foram incluídos no estudo alterações através de anotações e de mudanças de estruturas de classes; e (iv) não foi realizado o tratamento de falsos conflitos, como por exemplo, a modificação de dois atributos para um mesmo valor de inicialização, ou duas evoluções de métodos que tem a mesma AST de implementação.

O estudo empírico conduzido nesta tese foi realizado através de experimentos controlados e possui algumas ameaças a sua validade. Como ameaça interna pode-se destacar a confiabilidade dos dados contidos nos sistemas de gestão de tarefas da UFRN e SIG Software. A pesquisa considerou que os conflitos encontrados, a categorização do tipo de tarefa, do módulo afetado e dos artefatos estavam corretos. Para lidar com tal ameaça, um subconjunto de tarefas e conflitos levantados pela ferramenta apresentada foi validada manualmente, tem sido demonstrado a acurácia dos dados. Como ameaça de conclusão pode-se destacar que o total de quatro experimentos são insuficientes para permitir que o resultado seja extrapolado para toda a realidade dos sistemas SIG, sendo necessário a inclusão de outros pares de evoluções de sistemas para promover análises estatísticas. Vale ressaltar, entretanto, que os dados obtidos podem já ser úteis para guiar gerentes e desenvolvedores de software a lidar com os conflitos naqueles cenários específicos. Como ameaça externa à validade do estudo destaca-se que o fato de não ter sido considerada outras linhas de produtos Web não é possível considerar os resultados

obtidos para sistemas pertencentes ao mesmo domínio, nem tampouco a outros domínios existentes.

6.6 Discussões e lições aprendidas

Esta seção discute os resultados e lições aprendidas do estudo empírico realizado neste capítulo.

Resolução de Conflitos: o estudo mostrou que a maioria dos conflitos de merge de código podem ser resolvidos de forma automatizada ou semi-automatizada. A ferramenta de mineração revelou que no total de 6112 conflitos encontrados na comparação das 4 linhas clonadas: (i) cerca de 97% deles podem ser resolvidos automaticamente (10%) ou semi-automaticamente (87%); (ii) apenas 3% deles são conflitos diretos que exigem intervenção do engenheiro de software para serem resolvidos. Sob a perspectiva da tarefa, o estudo revelou que 71% delas não apresentam nenhum tipo de conflito, 9% apresentam conflitos que podem ser resolvidos automaticamente e 15% requer teste após integração semi-automatizada, apenas 15% necessitam de intervenção manual. Estes resultados preliminares indicam os benefícios da abordagem de merge e resolução proposta nesta tese.

Melhorias na modularidade: a análise de modularidade realizada a partir dos experimentos do estudo identificaram possibilidades de melhoria de usabilidade nos sistemas Web clonados. A análise de camadas, por exemplo, identificou que muitos dos conflitos que ocorrem nas classes de controle da camada Web (*Managed Beans*) podem ser evitados com a refatoração destas classes usando a padrão *Extract Method* que isola o comportamento funcional e coesivo em outros métodos, e com o padrão de projeto *Template Method* para tratar variáveis e comportamentos comuns nas classes existentes. Adicionalmente, o estudo também permitiu observar que o padrão de execução condicional usado para modularizar variabilidades de granularidade fina manifestam mais conflitos quando comparado com a abordagem composicional dos padrões de projeto. Isto representa uma evidência inicial dos benefícios das técnicas composicionais na evolução de sistemas web clonados, que podem ser investidos melhor em estudos futuros.

Colaboração de times: também foi possível observar no estudo que a identificação de conflitos podem ajudar em decisões tomadas pelos gerentes de projetos e líderes de times no contexto de sistemas clonados. A análise dos conflitos pode ser, por exemplo, processada por vários clones de um sistema para entender suas causas principais e tomar ações para reduzi-la. No estudo deste capítulo identificou-se que as tarefas de resolução de *bugs* são responsáveis por um alto percentual do total de conflitos (cerca de 60%). Testes conflitos podem ser reduzidos a partir da evolução apenas em módulos estáveis nos sistemas clonados, e, quando possível, corrigir problemas diretamente na *source* da linha clonada e não no *target*. Estratégia similar pode ser usada para reduzir conflitos nas camadas do sistema e nos núcleo/variabilidades. Estas hipóteses podem ser investigadas em trabalhos futuros.

6.7 Sumário

Ao longo deste capítulo foram apresentados os dados e as conclusões sobre o estudo empírico realizado para responder as questões de pesquisa: QP1 – Que tipos de conflitos de código ocorrem durante a evolução e *merge* da LPS; QP2 – Que tipos de tarefas – correção de *bugs*, novas funcionalidades, aprimoramentos – trazem mais conflitos no núcleo e variabilidade da linha de produto, QP3 – Onde e como os conflito ocorrem ao longo das camadas da LPS e a QP4 – Onde e como os conflitos ocorrem ao longo do *core*/variabilidades da LPS. Ao longo deste capítulo cada uma destas questões foi explorada e identificou-se que no *source* 5,5% das operações atômicas apresentaram algum tipo de conflito, já no *target* o percentual foi de 33,7%. Considerando a análise por tarefa, 28,7% das tarefas do *source* e 69,5% das tarefas do *target* possuem algum tipo de conflito. Estes números indicam que apesar da flexibilidade gerada pela clonagem é preciso estar preparado para tratar o alto número de conflitos que pode ser gerado em LPS clones durante a reconciliação. As tarefas de melhoria de funcionalidades foram as que geraram mais conflitos no *source* com 64,8%, já no *target* foram as de correção de *bugs* com 49,8%. A diminuição da necessidade da correção de *bugs* no clone pode reduzir consideralmente o total de conflitos. A análise de conflitos

por camada arquitetural demonstrou que a camada Web é onde ocorre a maioria dos conflitos de todos os tipos nas LPS analisadas, com 44% do geral. No que se refere a infraestrutura da LPS verificou-se que 0,6% dos conflitos ocorrem em variabilidades considerando as variabilidades explícitas. Já se considerarmos as variabilidades implíticas este percentual sobe para 26,35%. No que diz respeito a resolução de conflitos, nosso estudo identificou que 10,1% deles podem ser resolvidos de forma automatizada pela abordagem, 86,7% dos conflitos podem ser resolvidos de forma semi-automatizada e apenas 3,3% de forma manual pelo usuário. Considerando a sumarização por tarefas, os resultados do nosso estudo mostraram que 71,3% podem ser aplicadas automaticamente devido a identificação da ausência de conflitos, 8,9% podem ser resolvidas de forma automatizada, 14,6% de forma semi-automatizada e apenas 5,2% das tarefas com resolução manual.

7 Trabalhos Relacionados

Este capítulo apresenta e discute os trabalhos relacionados com esta tese de doutorado. Os trabalhos são organizados em três tipos: gerenciamento e tratamento de conflitos de código (Seção 7.1), desenvolvimento colaborativo (Seção 7.2), e abordagens e estudos relacionados à clonagem de linhas de produtos (Seção 7.3).

7.1 Gerenciamento e Tratamento de Conflitos de Código

Esta seção apresenta e discute trabalhos de pesquisa relacionados com o problema de gerenciamento e tratamento de conflitos quando durante a evolução de sistemas de software em equipes.

7.1.1 Abordagem para Detecção Preliminar de Colaborações e Riscos

Em Brun et al (Brun et al., 2013) os autores desenvolveram uma abordagem especulativa para buscar antecipadamente os conflitos, enquanto os desenvolvedores estão criando os seus códigos em cópias locais do espaços de trabalho. A abordagem considera dois tipos de conflitos: (i) conflitos textuais que são aqueles que ocorrem na mesmo artefato de código fonte; e (ii) conflitos de alta ordem (*high order)* que representam aqueles encontrados quando as contribuições são publicadas e tornam o código semanticamente incorreto, através de erros de compilação, falhas de testes ou outros problemas semânticos. O trabalho argumenta que se tais erros semânticos forem identificados antecipadamente pode ser mais fácil e menos custoso corrigi-los do que se eles forem propagados no código.

Para identificar os conflitos, os repositórios são comparados e quatro situações são apontadas:

- *Conflito Textual:* A comparação entre os repositórios identifica mudanças nos códigos fonte que não podem ser resolvidos automaticamente pelo sistema de controle de versões.

- *Falha em Build:* Os repositórios podem passar pelo *merge* automatizado do sistema de controle de versões mas, o resultado é uma falha de compilação.
- *Falha de Teste:* Os repositórios podem passar pelo *merge* automatizado mas, o resultado é uma falha nos testes automatizados.
- *Aceitação de Testes:* Os repositórios podem passar pelo *merge* automatizado e os testes automatizados executam com sucesso.

A abordagem conta com uma ferramenta de suporte para a análise especulativa, denominada *Crystal*. A ferramenta permite identificar a ocorrência de conflitos textuais e de alta ordem através da execução em períodos pré-determinados. Ela funciona tanto com sistemas de controle de versão centralizados (por exemplo: CVS e SVN) como para distribuídos (por exemplo: Git e Mercurial). Como principal contribuição do trabalho foram analisadas evoluções de nove sistemas *open source* para identificar os seus conflitos. Como resultado identificou-se que os conflitos textuais ocorrem em média em 16% das publicações no repositório de código. Os outros 84% do código não possuem conflitos textuais. A análise dos conflitos de alta ordem foi realizada em três sistemas (Git, Perl5 e Voldemort) que possuem testes unitários disponíveis e então os seguintes resultados foram obtidos:

- 69,6% dos merges tiveram aceitação dos testes;
- 14% tiveram conflitos textuais;
- 4,7% resultaram em falha de *build*;
- 11,7% resultaram em falha de teste;

A análise proposta do trabalho (Brun et al., 2013) identificou que cerca de 16% das publicações em repositórios possuem conflitos textuais. Os conflitos textuais identificados por aquele trabalho de pesquisa são equivalentes aos conflitos léxicos (diretos e textuais) deste trabalho, cujo percentual obtido foi de 13,4%. Nas situações de falha de *build* não há valores correspondentes no nosso trabalho pois sempre foram avaliadas *releases* sem falhas de compilação.

A abordagem proposta por Brun et al é focada na análise antecipada para buscar evitar os conflitos, ela pode ser usada de forma complementar ao *merge engine* proposto neste trabalho para identificar se o resultado do *merge* resultou em conflitos de alta

ordem (falha de *build* ou falha de teste). É também possível utilizar a abordagem para identificar antecipadamente os conflitos entre o *source* e o *target,* e negociar sua resolução antes da publicação das versões. A identificação de conflitos de alta ordem como falhas de *build* e testes é um ponto não suportado pela nossa abordagem e que pode ser tratado como trabalho futuro. Pode-se destacar também várias diferenças existentes entre os trabalhos de pesquisa: (i) no trabalho daqueles autores o *merge* utilizado é o oferecido nativamente pelos sistemas de controle de versões. A proposta desta tese possui *merge engine* próprio que resolve automaticamente uma parcela dos conflitos textuais relatados no trabalho de Brun et al através da analise semântica; (ii) diferentemente do trabalho desta tese, os autores não levam em consideração informações extraídas de mineração de repositório de software para identificação e análise dos conflitos e, dessa forma, também não oferecem suporte para computação de conflitos por tarefas; e (iii) no que diz respeito aos conflitos semânticos, a abordagem proposta nesta tese pode indicar métodos que devem ser re-testados, em contraposição a abordagem de Brun et al que confia e depende da cobertura dos testes automatizados.

7.1.2 Detecção Antecipada de Conflitos de Merge de Código

Guimarães & Silva (Guimarães & Silva. 2012) propõem uma abordagem de *merge* contínua que analisa os códigos publicados e não publicados em espaços de trabalhos locais para criar uma versão reconciliada do sistema. Esta versão é analisada, compilada e testada em segundo plano para identificar conflitos e problemas antes do desenvolvedor publicar seu código. A abordagem constrói uma árvore de elementos da linguagem (pacotes, classes, atributos e métodos) mais simplificada do que a árvore de sintaxe abstrata (AST- *Abstract Sintax Tree*) e realiza a comparação das versões locais dos desenvolvedores e a versão remota (publicada no repositório) das árvores de elementos. Com base em tal comparação detecta onde os conflitos estão ocorrendo em tempo de desenvolvimento. Os conflitos deste trabalho são categorizados nos seguintes tipos:

- **Conflitos estruturais:** conflitos encontrados durante o *merge* executado em segundo plano a partir da análise de árvore de mudanças. Estes conflitos são subdivididos em:

        - *Pseudo conflito direto:* ocorre quando diferentes elementos de um mesmo nó da árvore são modificados ou quando são alterados para um mesmo valor. Eles são equivalentes aos conflitos textuais da nossa abordagem.
        - *Conflito de mudança de atributos e métodos:* ocorre quando o mesmo atributo/método é alterado. Equivalente ao conflito direto da nossa abordagem.
        - *Conflito de mudança e remoção de nó:* ocorre quando um desenvolver altera o método e outro o deleta. Equivalente ao conflito direto da nossa abordagem.
        - *Conflito de tipo de nó inconsistente:* ocorre na tentativa de adicionar nós de diferentes tipos em uma mesma localização.
    - **Conflitos de linguagem:** após o *merge* realizado em segundo plano e a versão conciliada compilada, os erros de compilação decorrentes são denominados de conflitos de linguagem.
    - **Conflitos comportamentais:** são os conflitos que ocorrem no grafo de chamadas do elemento que está sendo alterado. A abordagem analisa as dependências do nó que foi modificado e verifica se no caminho não há mudanças que conflitem com a que foi realizada no nó.
    - **Conflitos de teste:** representa uma falha de teste ocorrida após a atualização e execução do sistema de *merge* e da execução dos testes unitários automatizados.

A abordagem introduz um *plugin* na ferramenta Eclipse que objetiva trazer a experiência do *merge* contínuo, de forma similar à compilação contínua já existente nesta ferramenta. Os conflitos são reportados ao time de desenvolvimento através de uma visão da IDE, alertando-os sobre conflitos existentes ou potenciais conflitos que podem ocorrer. Esta análise é feita avaliando o código local do desenvolvedor e comparando-o com o código remoto publicado.

O estudo empírico conduzido por Guimarães & Silva (Guimarães & Silva, 2012) não apresenta percentuais de conflitos que permita a comparação com os resultados obtidos neste trabalho. Porém, é possível identificar similaridades, utilizações complementares e diferenças da sua abordagem com a proposta neste trabalho. Dentre as similaridade ou utilizações complementares pode-se destacar: (i) a abordagem de

Guimarães & Silva analisa os conflitos através de uma árvore abstrata própria e computa os conflitos a partir dela. A nossa abordagem calcula os conflitos através da AST, do modelo de mineração do repositório e das informações obtidas no sistema de gerenciamento de mudanças. Ambas as abordagens utilizam análises semânticas para identificar os conflitos; (ii) os conflitos comportamentais propostos pelos autores são similares aos conflitos indiretos da nossa abordagem; (iii) os pseudo-conflitos estruturais são equivalentes aos conflitos textuais da nossa abordagem; e (iv) os demais conflitos estruturais (*mudança de atributos e métodos, mudança e remoção de nó e tipo de nó inconsistente*) são equivalentes aos conflitos diretos da nossa abordagem.

Dentre as principais diferenças destacam-se: (i) a abordagem de Guimarães & Silva define algoritmos para resolução de tipos específicos de conflitos. Por exemplo, no caso da remoção de métodos por um desenvolvedor e evolução por outro, o algoritmo de *merge* automaticamente descarta a remoção do método. Isto ocorre pois trata-se da resolução de conflitos sob a ótica da evolução de um mesmo produto, já na nossa abordagem o fato de um método ser deletado no *target* mas ser evoluído no *source* não deve ter sua resolução automatizada, pois a remoção pode ser necessária ao *target*; (ii) A abordagem proposta por eles é dirigida por publicações no repositório e não por tarefas. A análise por tarefa facilita a integração pois uma determinada tarefa pode conter mais de uma publicação no repositório, facilitando assim a integração de todas as publicações envolvidas com aquele contexto de mudança; (iii) a nossa abordagem utiliza a AST (*Abstract Syntax Tree*) para realização do *merge*, ao passo que a abordagem de Guimarães & Silva utiliza uma árvore mais simplificada. Apesar da árvore simplificada poder trazer vantagens no desempenho da análise de dependência, ela pode ser um fator limitador para a expansão da análise de conflitos para algoritmos mais avançados de *merge* semântico em granularidade fina, como por exemplo, análise em trechos de códigos.

7.1.3 Abordagem para Análise de Publicações Seguras em Repositório

Wocla et al (Wloka et al, 2009) apresentam uma abordagem para analisar o *software* em desenvolvimento e identificar mudanças que são passíveis de publicação no repositório de código com foco na análise dirigida por testes. O algoritmo proposto

pela abordagem busca flexibilizar as políticas de publicação no repositório de código-fonte (*commit policies*) garantindo que os desenvolvedores só possam publicar seus códigos quando todos os testes automatizados estiverem sem falhas. Os autores partem da premissa que quanto mais cedo as mudanças forem publicadas, menores serão as chances dos conflitos ocorrerem.

A abordagem proposta considera três tipos de políticas de publicação no repositório: restritiva, moderada e permissiva. A restritiva permite que apenas mudanças com todos os testes aceitos localmente sejam publicadas. A política moderada permite que mudanças sejam publicadas, desde que não falhe nenhum teste já previamente existente. Já a permissiva permite que as mudanças sejam publicadas mesmo com erros de testes, restringindo apenas à inexistência de erros de compilação.

Para determinar se uma determinada mudança é segura para publicação, a abordagem cria um modelo de mudanças atômicas (adição, remoção e atualização de atributos e métodos) e os relaciona com o grafo de chamadas obtido através da execução dos testes unitários. Vale ressaltar que tal estratégia é diferente da abordagem de Guimarães & Silva (Guimarães & Silva, 2012) pois ela não computa o grafo de chamadas obtido do método modificado no código fonte, e sim considera os testes unitários do sistema. Através do algoritmo de análise da associação entre o grafo de chamadas do teste unitário e as mudanças atômicas realizadas no código determinam-se quais mudanças são seguras para serem publicadas no repositório.

Como resultado de uma avaliação preliminar da abordagem o trabalho apresenta os seguintes dados:

- 4,6% das mudanças são seguras para serem publicadas de acordo com a política restritiva, ou seja, não geram nenhum erro durante a execução dos testes já existentes e novos;
- 31,4% das mudanças são seguras para serem publicadas de acordo com a política moderada, ou seja, novos testes podem ter falhas mas os já existentes devem ser aceitos;
- 99,5% das mudanças são seguras para serem publicadas de acordo com a política permissiva, ou seja, que não geram erros de compilação.

A abordagem proposta por tais autores é centrada na publicação de contribuições de código seguras no repositório, ao invés de análise de conflitos. Uma publicação segura através da análise dos testes unitários como proposto por tal abordagem pode ser visto como uma forma alternativa de computar conflitos indiretos.

Dentre as principais diferenças destacam-se: (i) a análise da dependência do grafo de chamadas é realizada a partir dos testes unitários, a abordagem desta tese realiza uma análise de dependências partindo dos elementos que foram modificados. A análise dirigida por testes unitários é capaz de cobrir apenas os códigos chamados pelos testes, ou seja, caso um determinado código seja alterado e não tenha um teste associado seu impacto não será detectado. A análise por dependência de código utilizada na abordagem desta tese analisa as dependências entre elementos de código de forma mais completa, tendo assim um maior custo; (ii) a abordagem não é dirigida por tarefas e sim por revisões de código no repositório, a análise por tarefa pode facilitar a integração por possibilitar que as diferentes revisões relacionadas a ela, possam ser integradas como uma única unidade atômica, cuja atualização se faz necessária; e (iii) não é possível comparar os resultados obtidos para os estudos empíricos conduzidos por aqueles autores, porque os conflitos abordado por eles, representam publicações (*commits*) seguras de acordo com uma determinada política.

7.1.4 Categorização e Estudos de *Merge* de Código

Em Mens (Mens, 2002) o autor destaca várias opções de *merges* disponíveis na literatura, categorizando-os em: (i) merge bidirecional ou de três vias; (ii) merge textual, sintático, semântico ou estrutural; e (iii) merge baseado em estado, em mudanças ou em operador. O *merge* bidirecional – critério (i) – compara duas versões sem levar em conta o ancestral que origina ambas as versões; o *merge* de três vias considera as duas versões e o ancestral, sendo mais robusto uma vez que inclui mais uma versão na comparação. O *merge* textual – critério (ii) – considera os artefatos como simples arquivos texto, sem entender sua semântica. O *merge* sintático entende a gramática da linguagem e é capaz de resolver conflitos simples como modificações concorrentes em comentários ou mesmo a introdução de espaços e tabulações na classe. O *merge* semântico compreende os elementos da linguagem construindo árvores de elementos, como a AST ou outras. O *merge* estrutural avalia a mudança da estrutura da classe, como heranças, interfaces e polimorfismo. É o tipo mais complexo de *merge* e

normalmente não possui suporte ferramental por não possuir todas as informações necessárias no código fonte. No merge baseado em estados – critério (iii) – apenas a informação da versão original e suas revisões anteriores são usadas. O merge baseado em mudança considera precisamente as informações que foram feitas no software apenas durante sua evolução. O merge baseado em operadores é um caso particular do baseado em mudanças onde as mudanças são representadas em modelos bem formados. A abordagem desta tese pode ser considerada de acordo com a categorização proposta em (Mens, 2002), como: (i) um merge em três vias já que ela considera as informações de ancestralidade; (ii) um merge semântico já que faz uso de informações disponíveis na árvores de elementos da AST; e (iii) merge baseado em operador já que representa as mudanças através de um modelo de transformação.

Zimmerman (Zimmermann, 2007) avaliou a evolução de quatro sistemas de código aberto (JBoss, JEdit, GCC e Python) que utilizam o gerenciador de versão CVS e concluiu em seu estudo empírico que os conflitos léxicos em tais projetos, ocorreram entre 23-46% dos cenários de *merge*. Apel et al (Apel et al, 2011) analisam o impacto dos sistemas de controle de versões não estruturados e estruturados na resolução de conflitos de código. Os sistemas não estruturados são os baseados em texto, ou seja, registram a revisão como incrementos textuais sem reconhecer a semântica do seu conteúdo, os sistemas tradicionais como o CVS, SVN e Git pertencem a esta categoria. Os sistemas estruturados são específicos de linguagem e reconhecem a semântica dos elementos que estão sendo publicados através da AST da classe em questão. Apel et al sugerem uma abordagem intermediária denominada *merge* semiestruturado. Nesta abordagem, os artefatos são representados através de uma árvore de elementos da linguagem e a partir dela realizam-se as comparações do *merge*. Um dos diferenciais da abordagem semiestruturada é que ela permite configurar gramáticas para diferentes linguagens. Os conflitos são categorizados em: (i) conflitos de ordenação – aqueles que podem ser resolvidos simplesmente inserindo os elementos da linguagem na classe destino através da análise da ordem de elementos; e (ii) conflitos semânticos – aqueles que exigem um algoritmo específico de tratamento para resolvê-los, por exemplo: a adição de implementação de uma interface na classe deve checar se todos os métodos foram implementados. A abordagem proposta pelos autores suporta as gramáticas das linguagens Java, C# e Python, apesar de outras poderem ser construídas. O estudo empírico demonstrou que a comparação com a abordagem não estruturada o número de

conflitos é diminuído em cerca de 60%, o número de linhas de código com conflito reduziu 82% e o número de artefatos conflitantes reduziu 72%. O trabalho de Apel et al relaciona-se com a nossa abordagem pelo tratamento semiestruturado das informações de evolução, identificando assim os conflitos mais precisamente. O trabalho deles não lida com conflitos indiretos, já que o objetivo principal é realizar merge semântico e diminuir conflitos apenas léxicos. Em termos de números obtidos o estudo empírico obteve 60% de resolução dos conflitos dos sistemas investigados enquanto a nossa abordagem resolve 96,8% dos conflitos identificados (incluindo indiretos).

Além de pesquisas acadêmicas sobre *merge* semântico também é possível encontrar ferramentas industriais disponíveis, como por exemplo, a *Semantic Merge* (Codice Software, 2013). Esta ferramenta se assemelha ao *merge engine* da nossa abordagem já que ambas fazem análise dos elementos da linguagem e são capazes de realizar o merge semântico, ou seja, aquele que reconhece os elementos da linguagem. No entanto, a ferramenta *Semantic Merge* por ser de propósito comercial possui características de usabilidade importantes não implementadas em nossa abordagem, tais como: rastreabilidade ao renomear, mover e deletar artefatos em *refactoring* e *merge* visual. A ferramenta proposta não oferece suporte para *merge* orientado à tarefa nem análise de conflitos indiretos, o que é suportado pela nossa abordagem. A criação de uma versão industrial da abordagem é um trabalho futuro de continuidade desta pesquisa.

7.2 Ferramentas e Metodologias de Desenvolvimento Colaborativo

Com o objetivo de evitar conflitos durante o desenvolvimento, diversas pesquisas estão sendo desenvolvidas com o foco na proposição de metodologias e ferramentas que possam tornar o desenvolvimento consciente (*awareness development*). Esta metodologia de desenvolvimento de software permite que o desenvolvedor possa entender as atividades sendo realizadas por outros desenvolvedores, proporcionando assim um contexto para suas próprias atividades (Dourish & Bellotti, 1992).

Em Sarna et al (Sarma et al, 2003), os autores propõem uma ferramenta para auxiliar no desenvolvimento colaborativo, denominada *Palantir*. Através do uso desta

ferramenta os desenvolvedores podem identificar mudanças realizadas por outros desenvolvedores e comunicar-se para tratar o possível conflito antes que ele aconteça. *Palantir* possui três principais características: (i) é um mecanismo de coordenação baseado mais no *workspace* do ambiente de desenvolvimento do que no repositório; (ii) ele continuamente informa ao desenvolvedor o que os outros estão fazendo; e (iii) fornece uma visão geral dos outros *workspaces* para detectar tanto conflitos diretos como indiretos. Nesta abordagem os conflitos diretos são aqueles cujo desenvolvedores modificam o mesmo artefato. Já os conflitos indiretos são os que geram erros de linguagem por modificações de outros membros, por exemplo, uma mudança de assinatura de um método no *workspace* que gera erros de compilação em outra classes na equipe. O estudo empírico desenvolvido em Sarna et al (Sarna et al, 2012) identificou que o uso do *Palantir*: (i) encontra um alto número de conflitos quando comparado a não utilização da ferramenta; (ii) resultou em menos conflitos que são resolvidos em código; e (iii) possui um esforço extra aceitável no processo de desenvolvimento. Como limitação destaca-se a impossibilidade da análise do conflito indireto pelo grafo de chamadas, identificado como trabalho futuro em Sarna et al (Sarna et al, 2012).

Em Dewan & Hegde (Dewan & Hegde, 2007), a ferramenta CollabVS é proposta para melhorar o suporte ferramental para o desenvolvimento colaborativo. Esta ferramenta identifica conflitos: (i) diretos – aqueles que realizam modificação concorrente no mesmo artefato; e (ii) indiretos – que percorrem o grafo de dependências da mudança em questão. A ferramenta fornece canais de comunicação, tais como *chats* e videoconferência, para que os desenvolvedores possam se comunicar e resolver os conflitos de código mais rapidamente. Quando dois desenvolvedores estão editando o mesmo artefato, o CollabVS identifica a semântica da mudança (classe, método ou atributo), assim como notifica e fornece canais de comunicações para que os membros envolvidos naquele conflito possam utilizar durante sua resolução. A ferramenta CollabVS foi desenvolvida para a linguagem Microsoft C#.

Mais recentemente, Hattori & Lanza (Hattori & Lanza, 2010) abordam o problema de filtrar informações do desenvolvimento colaborativo de forma a apresentar apenas mudanças relevantes para um dado desenvolvedor, evitando assim que o mesmo perca tempo com mudanças não relacionadas ao contexto do seu trabalho. Eles propõem a ferramenta *Syde*, um *plugin* da IDE Eclipse, que fornece diversas visualizações para

exibição dos conflitos e realização do *merge* quando eles acontecem. São quatro aplicações para: (i) fornecer informações relevantes aos desenvolvedores (*Scamp*); (ii) fornecer alertas de conflitos potenciais (*Conflicts*); exibir as informações de desenvolvimento e conflitos de forma gráfica (*Manhattan*); e permitir analisar o histórico das sessões de desenvolvimento (*Replay*) (Hattori, 2012). O *Syde* é capaz de identificar conflitos estruturais (diretos e textuais) e analisar o grafo de chamadas para buscar conflitos nas dependências. Em sua tese de doutorado, Hattori (Hattori, 2012) conduz diversos estudos empíricos para avaliar cada uma das aplicações (*Scamp, Conflicts, Manhatan e Replay*). Os estudos não possuem dados percentuais para comparar com os nossos resultados, eles se foca em avaliações qualitativas do uso da ferramenta e como ela contribui para o processo de desenvolvimento.

De forma geral, as pesquisas realizadas na proposição de ferramentas de desenvolvimento colaborativo têm como foco principal o desenvolvimento consciente e a prevenção de conflitos. Elas podem ser vistas como complementares a abordagem de identificação e resolução de conflitos proposta nesta tese, podendo ser utilizadas no ambiente de desenvolvimento como forma de diminuir os conflitos. As equipes de manutenção das LPS clonadas podem, por exemplo, utilizá-las para detectar antecipadamente os possíveis conflitos. No entanto, tal prática é de difícil implementação, pois as equipes trabalham em cenários de requisitos de clientes distintos, o que dificulta a comunicação no tratamento dos conflitos por cada um conhecer apenas o seu cenário. Uma possibilidade que poderia ser explorada é a equipe de manutenção da infraestrutura da LPS usar as informações relacionadas a tais conflitos para indicar e antecipar possíveis problemas futuros de integração de versões clonadas das LPS. De forma geral, as abordagens apresentadas não se propõem a resolver conflitos, mas apenas identificá-los e fornecer mecanismos para que as próprias equipes resolvam de forma manual. Não há suporte a *merge* das mudanças entre os membros do time, esta tarefa é delegada ao sistema gerenciador de versões.

7.3 Abordagens para Gerenciamento de Produtos Clonados em LPS

Dentre as várias técnicas de reuso de código preconizadas pela literatura a clonagem é uma das menos recomendadas devido à replicação de artefatos similares que podem trazer problemas de manutenção quando evoluídos por equipes independentes. Entretanto, a abordagem de reutilização por clonagem tem sido bastante utilizada na indústria permitindo a derivação de novos produtos através da criação de cópia de um produto existente e da customização de funcionalidades específicas que atendam clientes existentes. Tal abordagem de reuso é denominada "*clone-and-own*" (Rubin, J. et al, 2012).

A utilização da clonagem também tem seus desafios conhecidos, dentre os quais destaca-se o gerenciamento e a rastreabilidade das mudanças que são feitas em cada um dos clones. Rubin et al (Rubin et al, 2012) propuseram uma abordagem para capturar as informações necessárias para o gerenciamento de linhas de produto clonadas, denominada de PL-CDM: *Product Line Changeset Dependency Model*. A abordagem extrai de sistemas de controle de versões informações sobre os artefatos modificados e de sistemas de gestão de tarefas as informações sobre as características afetadas. Com estas informações extraídas realiza outras duas operações: (i) agregação de característica (*feature agregation*) e (ii) cálculo de dependência entre características (*feature dependency*). A agregação de características agrupa de forma incremental o código em conceitos de alto nível de engenharia de LPS. O cálculo de dependência entre características realiza uma analise estática no código da aplicação para identificar sua interdependência e, consequentemente, as dependências entre as características da LPS que utilizam aqueles códigos. A abordagem PL-CDM computa a dependência entre características, uma funcionalidade que não foi explorada e implementada na nossa abordagem, sendo caracterizada como trabalho futuro. Apesar de bastante interessante, tal abordagem não foi ainda completamente implementada nem tampouco avaliada no contexto de engenharia de LPS clonadas.

Um outro trabalho relacionado à clonagem é o estudo do uso da técnica de clonagem de LPS na indústria de software, Dubinsky et al (Dubinsky et al, 2013) investigaram o uso da abordagem da clonagem de produtos em seis grandes empresas das áreas de armazenamento de dados, aeroespacial/defesa e automotiva. O objetivo da pesquisa era identificar se a abordagem atende as necessidades da indústria, bem como

verificar os benefícios e desvantagens da sua utilização. Para obtenção dos resultados foram aplicados questionários nestas empresas e a partir de então foram identificadas quatro principais características da prática da clonagem:

- **Eficiência:** A clonagem é considerada um mecanismo simples e eficiente de reutilização pois permite economiza de tempo e recursos. Ela possibilita iniciar rapidamente o desenvolvimento de um novo produto a partir de um conjunto de artefatos previamente testados e considerados estáveis. Ao mesmo tempo, a utilização da técnica proporciona independência e liberdade para mudar os artefatos do clone conforme a necessidade;
- ***Overhead*:** A manutenção de artefatos clonados envolve um custo adicional, pois elas precisam ser realizadas cada um dos artefatos das cópias criadas. Desta forma, a propagação das modificações também não é uma tarefa trivial. Por exemplo, uma correção de um *bug* em uma LPS origem de vários clones pode exigir a replicação desta correção em cada um dos clones;
- **Pensamento de curto prazo:** A falta de recursos para investimentos em estratégias sistematizadas de reuso, bem como a falta de abordagens conscientes de reuso, conduzem à escolha da clonagem como uma das práticas favoritas. As organizações frequentemente focam no sucesso dos seus produtos individuais postergando preocupações de reuso para o futuro.
- **(Falta de) Governança:** Uma base de conhecimento sobre reuso é raramente mantida. O reuso não é medido e frequentemente não existem regras ou responsabilidades nos processos de desenvolvimento e práticas sistematizadas de reuso.

A pesquisa conclui que apesar de haver desvantagens os engenheiros de tais empresas estão satisfeitos com a prática da clonagem e pretendem continuar adotando-a. Apesar de trabalhos de pesquisa (Riva & Rosso, 2003) (Bosch, 2002) identificarem a clonagem como o menor nível de maturidade em engenharia de linhas de produto de software, o estudo revela que organizações importantes e com processos maduros utilizam esta prática. Neste contexto, a abordagem de detecção de conflitos proposta nesta dissertação oferece um caminho possível para lidar com o desenvolvimento e evolução de LPS clonadas.

A rapidez inicial de resultados atrai muitos utilizadores para a clonagem, no entanto, o crescimento do número do clones conduz a custos adicionais para realizar o gerenciamento das modificações. Neste contexto, Rubin & Chechik (Rubin & Chechik, 2013) propuseram um *framework* para o gerenciamento de produtos clonados. O *framework* define sete operações conceituais para reconciliação de produtos. As operações são listadas abaixo e ilustradas na Figura 7-1.

- ***findFD:*** Retorna um conjunto de especificações das características de um determinado produto variante;
- ***findFI:*** conhecida como *feature location*, retorna os artefatos relacionados com uma determinada característica;
- ***dependsOn?:*** operador que analisa a dependência entre duas características para decidir se a aplicação da característica f1 requer a aplicação da f2, por exemplo.
- ***Same?:*** identifica se duas características de produtos distintos são idênticas ou não;
- ***interact?:*** verifica se dois conjuntos de características de produtos distintos podem trabalhar juntas;
- ***compose (merge):*** unifica um conjunto de características de produtos distintos ou de um produto completo.
- ***reorganize*:** produto resultante do *refactoring,* seguindo a arquitetura da linha de produto.

	Operator	Input	Output
1.	`findFD`	*variant*	*set of FDs*
2.	`findFI`	*variant* × *FD* × *property*	*FI*
3.	`dependsOn?`	$<FD_1, variant> \times <FD_2, variant> \times property$	*set of witnesses*
4.	`same?`	$<FD_1, variant_1> \times <FD_2, variant_2> \times property$	*set of witnesses*
5.	`interact?`	*set of* $<FD_i, variant_i> \times property$	*set of witnesses*
6.	`compose`	$system_1 \times \ldots \times system_n \times matches \times resolution$	*system*
7.	`reorganize`	*system*	$system'$

Figura 7-1 Operadores para gerenciamento de produtos clonados variantes

Em Rubin et al (Rubin et al, 2013b) foi realizado um estudo empírico com três empresas para avaliar o uso do framework proposto e a utilização dos operadores. Em cada estudo de caso foi demonstrado a aplicabilidade (ou não) dos operadores para atividades de desenvolvimento tanto no cenário de migração para um processo de engenharia de LPS como na manutenção de produtos clonados. A conclusão do estudo empírico é que os operadores são aplicáveis ao cenário real, mas ainda necessitam de diversas especializações para tratar os casos específicos. Operadores como *merge* e *compare* foram propostos mas são ainda conceituais. A abordagem desta tese permite a comparação de mudanças realizadas em tarefas e artefatos de código, assim como quantifica os possíveis conflitos para a sua integração (*merge*). Uma diferença central entre a abordagem proposta nesta tese e os operadores propostos por Rubin et al, reside no fato deles serem mais orientados a características de LPS, enquanto nosso trabalho é mais orientado a tarefas de desenvolvimento (*issues*) tal como definido por sistemas de gerência de mudanças. O fato dela ser baseada em tarefas de desenvolvimento, permite que as mesmas sejam movidas e integradas entre *clones* de uma LPS, de forma similar a forma de trabalho de desenvolvedores quando realizando atividades de *merge* de código. Vale ressaltar que tarefas de desenvolvimento do tipo melhoria de funcionalidades existentes ou o desenvolvimento de novos casos de uso podem ser usados para caracterizar evolução ou criação de características da LPS, sendo possível, portanto, para a nossa abordagem trabalhar com tal conceito.

7.4 Sumário

Este capítulo apresentou e discutiu diversos trabalhos relacionados à análise de conflitos, *merge* em repositórios, desenvolvimento colaborativo e gerenciamento de abordagens de reuso através do *clone-and-own*. Identificou-se diversas pesquisas focadas na análise dos conflitos, principalmente visando tratá-los no decorrer do processo de desenvolvimento. Algumas ferramentas - como a *Palantir, ColabVS e Syde* – oferecem suporte ao desenvolvimento colaborativo fornecendo um canal de comunicação entre os desenvolvedores e implementando o desenvolvimento consciente. Estas ferramentas também detectam conflitos e avisam com antecedência para evitar que eles se manifestem na versão publicada do repositório. No que se refere as pesquisas de abordagens de *clone-and-own*, um dos principais focos deste trabalho, a

análise das pesquisas relacionadas demonstram que os resultados são ainda preliminares, tendo sido a abordagem sistematicamente combatida até recentemente. Porém, com seu uso sistemático na indústria, a área de pesquisa vem recebendo mais atenção por parte da comunidade. E, em especial, o desafio de migrar de abordagens baseada na clonagem para abordagens que promovam gerência de variabilidades efetiva.

A proposta desta tese aborda tanto o gerenciamento das evoluções dos clones para permitir sua reconciliação futura como também a análise, detecção e resolução de conflitos. Os trabalhos propostos por Rubin e outros (Rubin et al, 2013b) se relacionam com os objetivos de gerenciamento de evolução de clones, já os trabalhos anteriormente citados na análise e computação de conflitos se relacionam nas estratégias de detecção de conflitos. A proposta desta tese aborda os dois problemas de forma integrada em uma única abordagem, que é capaz de rastrear os manutenções realizadas nos clones e reconciliá-las com o suporte à análise e resolução automatizada de conflitos.

8 Conclusões

Este trabalho apresentou uma abordagem para reconciliação de linhas de produtos clonadas orientada a tarefas, assim como um estudo empírico de caracterização e análise da resolução de conflitos em clones das LPS SIG-UFRN usando a abordagem proposta. Para o desenvolvimento desta tese foram definidas quatro questões de pesquisa principais, as quais são discutidas a seguir.

A primeira questão de pesquisa definida para esta tese foi identificar se "**é possível evoluir concomitantemente LPS de sistemas de informação Web, através da técnica da clonagem e, posteriormente, reconciliar as tarefas de evolução de cada LPS de forma independente?**". Nosso trabalho propôs uma abordagem para reconciliar tarefas de evolução (Capítulo 4), assim como apresentou um estudo empírico (Capítulo 6) que demonstrou que é possível promover a resolução automatizada e semi-automatizada de um grande número de conflitos oriundos de tal reconciliação através do uso de técnicas de mineração de repositório e análise de código. A abordagem realiza inicialmente uma mineração das tarefas de desenvolvimento (*issues*) realizadas em sistemas de gestão de mudanças (*issue trackers*), assim como relacionando com as revisões de código publicadas no sistema de gerência de versões. A partir destes dados, a mineração gera um modelo de mudanças indicando as tarefas e elementos de código modificados para que o analisador de conflitos possa atuar. Com os conflitos identificados, os engenheiros responsáveis pelas LPS clonadas podem saber precisamente que tarefas podem ser reconciliadas com segurança, quais podem ser automaticamente ou semi-automaticamente resolvidas pela abordagem e quais delas deverão ser objeto de resolução manual. A reconciliação de versões e resolução de conflitos orientada a tarefas aproxima a abordagem aos modelos usuais de desenvolvimento podendo ser aplicada em cenários reais.

A segunda questão de pesquisa investigada foi identificar **"que tipos de conflitos ocorrem quando se está realizando a integração e *merge* de tarefas de desenvolvimento de linhas de produto de software clonadas, e em particular, de LPS de sistemas de informação Web?".** Buscando responder a tal questão de

pesquisa, foi conduzido um estudo empírico de análise de 4 evoluções de LPS clonadas de sistemas de informação da UFRN. O estudo identificou que 86,7% dos conflitos manifestados são indiretos, dos quais 39,2% no sentido *target-source* e 47,5% no sentido *source-target*. Este resultado indica o alto número de mudanças na LPS *source* que impactam indiretamente a LPS *target* (clone) na reconciliação. Além disso, 10,1% são conflitos textuais e 3,3% conflitos diretos, ou seja, os conflitos léxicos representam 13,4% de todos os conflitos. A análise por tarefa demonstrou que 28,7% da tarefas da LPS *source* manifestaram conflitos. Dentre estas, 30,9% são de conflitos textuais, 29,6% de indiretos, 21,2% indiretos e textuais, 7,5% de diretos e 10,8% de outras combinações de conflitos. Já na LPS *target*, 69% das tarefas manifestaram conflitos, dentre as quais 30,9% foram textuais, 29,6% indiretos, 21,2% indiretos e textuais e 18,3% com pelo menos um conflito direto. Nota-se que o impacto na LPS *clone* (target) é o mais relevante, pois quase 70% das tarefas desenvolvidas manifestaram algum tipo de conflito dentre os quais 60,9% das tarefas possuem pelo menos um conflito indireto, demonstrando o relevante impacto indireto gerado pela reconciliação do *source*. Em termos de operações atômicas os conflitos ocorrem normalmente em atualização métodos, representando 84,4% do *source* e 92,1% do *target*. No *source*, o tipo de tarefa que mais gerou todos os diferentes tipos de conflitos foi a de melhoria de funcionalidades, com 54,5% dos conflitos diretos, 62,2% dos textuais e 65,5% dos indiretos. No *target*, destaca-se a alta incidência de conflitos indiretos nas tarefas de correção de *bugs*, com 62,1%, indicando uma possível melhoria no processo de publicação de versões do *source* para diminuição do número de *bugs*.

A terceira questão de pesquisa estipulada foi analisar **"quais conflitos de código podem ser resolvidos de forma automatizada, semi-automatizada ou manual quando se está reconciliando LPS clonadas?"**. A partir do estudo empírico realizado concluiu-se que (i) 10,1% dos conflitos podem ser resolvidos de forma automatizada pela abordagem; (ii) 86,7% dos conflitos são indiretos e podem ser resolvidos de forma semi-automatizada; e (iii) apenas 3,3% dos conflitos são diretos e precisam ser resolvidos de forma manual pelo usuário. No geral, 96,8% dos conflitos encontrados podem ser resolvidos de forma automatizada ou semi-automatizada. No que tange a análise por tarefas verifica-se que (i) 71,3% das tarefas do *source* podem ser incorporadas no *target* com segurança uma vez que a análise não detectou nenhum

conflito nas mesmas; (ii) 8,9% das tarefas são resolvidas de forma automatizada; e (iii) 14,6% de forma semi-automatizada requerendo a realização de testes para a sua condução. Por fim, apenas 5,2% de todas as tarefas do *source* são passíveis de resolução manual por parte do usuário. Os resultados do estudo empírico conduzido demonstram a contribuição da abordagem na resolução automatizada ou semi-automatizada de um número significativo de conflitos (96,8%) e tarefas (94,8%), sem a necessidade de intervenção manual.

A quarta questão de pesquisa definida foi **"qual o impacto de tais conflitos de integração de código das LPS clonadas ao longo das suas camadas, assim como no núcleo e variabilidades das mesmas?"**. A partir do estudo empírico foi possível identificar que 44,4% dos conflitos no *source* e no *target* acontecem na camada Web, seguidas pelas camadas de negócio, acesso a dados, entidade e outros quando se tratam de conflitos léxicos (textuais e diretos). Em geral, as camadas mais conflitantes foram também as tiveram mais evoluções atômicas nas LPS investigadas no estudo. A camada Web também manifestou mais conflitos indiretos (45,1% dos conflitos indiretos), sendo seguida pela camada de Entidade (34,1%). Os resultados do estudo conduzido mostraram que a camada Web é a que exige mais atenção para adoção de técnicas que ajudam na resolução de conflitos, pois foi a que apresentou um maior percentual para todos os tipos de conflitos encontrados. Na média, 39% das tarefas do *source* e 48% do *target* possuem conflitos em apenas uma camada. Os conflitos diretos, que são resolvidos manualmente pelo desenvolvedor, são 49% manifestados somente em uma camada. No que tange a infraestrutura da LPS, as variabilidades explicitamente implementadas para as LPS investigadas (Capítulo 3) estavam relacionadas a: 0,4% da evoluções, 5,9% das tarefas e 2,2% dos conflitos. Os conflitos que ocorreram nas variabilidades foram 91,9% implementados usando a técnica de execução condicional. Quando o estudo considerou as variabilidades implícitas catalogadas (Seção 6.5.4) verificou-se que o número real de conflitos em variabilidades foi maior (26,35%), demonstrando a necessidade de explicitamente implementar tais variabilidades quando elas surgem, bem como manter a consistência das suas informações de mapeamento em código.

8.1 Contribuições da Tese

As principais contribuições desta tese são:

- Uma abordagem para evolução e reconciliação de LPS clonadas orientada a tarefas que integra técnicas de mineração de repositórios de software e análise de código para a resolução dos conflitos oriundos da reconciliação;
- Criação de uma ferramenta de apoio a abordagem com um *merge engine* que realiza o *merge* através da manipulação da AST (*Abstract Syntax Tree*) e de informações obtidas da mineração;
- Documentação de técnicas composicionais e anotativas usadas na modularização de variabilidades de LPS de sistemas de informação Web de larga escala;
- Condução de um estudo empírico de análise e caracterização de conflitos de reconciliação de LPS clonadas de sistemas de informação Web de larga escala que trouxe resultados e análises, tais como:
    - onde os conflitos se manifestam nas LPS clonadas considerando os pares de conflitos, operações atômicas e tarefas;
    - como os tipos de tarefas (correção de *bugs*, melhoria de funcionalidades e novos casos de usos) contribuem para a manifestação dos conflitos;
    - onde os conflitos ocorrem em termos de camadas arquiteturais e identificação de alternativas de implementação e boas práticas que possam reduzir tais conflitos;
    - a relação dos conflitos e evoluções com o núcleo e variabilidades da LPS sendo evoluída e clonada.

8.2 Limitações do Trabalho

As principais limitações do trabalho são:

- o estudo empírico conduzido analisou conflitos de reconciliação de código de um número baixo (quatro) de LPS clonadas. Dessa forma, embora os resultados obtidos no estudo já sejam úteis para direcionar políticas específicas para auxiliar na resolução de conflitos de LPS clonadas dos SIG da UFRN, ainda são

insuficientes para serem generalizados para outras LPS de sistemas de informação Web ou mesmo a rede de usuários dos sistemas SIG-UFRN;

- o estudo realizado não considerou diversos tipos de modificações que precisam ser considerados durante a análise de conflitos, tais como, páginas JavaServer Pages, modificações em metadados e relações de herança/implementação. A versão atual da ferramenta proposta já contempla implementação para análise de tais modificações que serão incluídas em trabalhos futuros;
- a versão atual da abordagem é limitada a linguagem Java, mas boa parte dos resultados reportados nesta tese podem ser estendidos para outras linguagens orientadas a objetos;
- o estudo conduzido neste trabalho considerou uma profundidade de nível seis na análise de conflitos indiretos. Versões futuras ou industriais da abordagem precisam ampliar tal análise para considerar políticas específicas de análise de tal profundidade, já que análise completa da profundidade de grafos de chamadas parece bastante complexo quando considerando sistemas de software de larga escala;

8.3 Trabalhos Futuros

Com o objetivo de suprir as limitações identificadas bem como dar continuidade à pesquisa desenvolvida nesta tese de doutorado, os seguintes trabalhos futuros foram identificados:

- *Condução de novos estudos empíricos:* o estudo empírico conduzido nesta tese pode ser replicado para considerar outras cenários de LPS clonadas dos sistemas SIG da UFRN, assim como ser aplicado a outras LPS ou sistemas de software comerciais ou *open-source*. Tais replicações permitirão entender melhor os conflitos que ocorrem durante a evolução e reconciliação de LPS ou sistemas de software clonados de um domínio específico, assim como permitir um tratamento estatístico dos resultados obtidos. Um outro estudo de interesse é analisar se as LPS clonadas que usam a técnica de implementação de variabilidades anotativa quando são refatoradas para utilizarem técnicas

composicionais podem ocasionar a geração de menos conflitos para a integração e resolução automática de conflitos.

- *Condução de um estudo do impacto da abordagem:* o estudo empírico conduzido nesta tese identificou que 96,8% dos conflitos e 94,8% das tarefas podem ser resolvidas de forma automatizada ou semi-automatizada, mostrando o potencial da abordagem. Porém, é preciso conduzir estudos de caso com desenvolvedores para avaliar a usabilidade e eficiência da abordagem no real auxílio de desenvolvimento e evolução de LPS clonadas. Neste sentido, além de avaliar sua eficiência, deve-se considerar o custo de utilização da abordagem versus o custo de realização de *merge* manual.

- *Analisar o impacto dos conflitos em termos de linhas de códigos modificadas:* um determinado conflito pode ocorrer em uma fração de duas linhas de código ou ao longo de várias linhas. O esforço de resolução manual neste caso será diferente de acordo com o total de linhas de código envolvidas. O modelo da abordagem proposta na tese oferece suporte para armazenamento do código modificado (denominado de *CodePiece*), porém como a abordagem não oferece suporte para sua análise ela não foi considerada na mineração. Um trabalho futuro será conduzir novos estudos ou refinar os existentes que quantifique as linhas de código relacionadas a um conflito direto.

- *Desenvolver uma ferramenta com características industriais para utilização corporativa:* a ferramenta proposta nesta tese foi implementada buscando responder questões de pesquisas específicas. A adoção industrial da mesma exige melhorias no suporte visual para operações de *merge*, na validação de diversas situações possíveis do *merge engine*, na criação de novos algoritmos de resolução de conflitos, e no suporte à mineração em outros repositórios como o Git e Mercurial, dentre outras melhorias.

- *Pré-computação dos conflitos indiretos:* utilizar frameworks de análise estática, tais como o WALA[4], para otimizar a computação da análise dos conflitos indiretos. Atualmente todo o calculo é feito por demanda utilizando o Eclipse

[4] http://wala.sourceforge.ne

JDT, isso traz um alto custo para o processamento dos conflitos indiretos. Tal custo traz limitação para execução *on-the-fly* da abordagem.

- *Analise detalhada dos casos de conflitos indiretos:* 86,7% dos conflitos encontrados no estudo conduzido são indiretos. Analisar em detalhes estes tipos de conflitos pode ser útil para determinar em maiores detalhes suas categorias e retirar algumas dependências fracas, como por exemplo, a de um método que evoluiu por que adicionou exceções *runtime*. Assim, a condução de um estudo de análise dos tipos de conflitos indiretos e que tipos de refatorações poderiam ser realizadas para diminuir o acoplamento e aumentar a modularidade também se apresenta como um trabalho futuro.

- *Promover integração com trabalhos relacionados:* alguns trabalhos apresentados no Capítulo 7 podem ser aplicados de forma complementar à nossa abordagem. Um trabalho futuro proposto seria promover estudos com a utilização de ferramentas de detecção antecipadas de conflitos ou utilizar as ferramentas de desenvolvimento colaborativo com o objetivo de diminuí-los. A integração com as abordagens existentes de gerenciamento de produtos clonados também pode ser avaliada. A integração com o modelo PL-CDM: *Product Line Changeset Dependency Model* como formato padrão poderia ser testada (Rubin et al, 2012). A integração entre as operações conceituais propostas por Rubin & Chechik (Rubin. J, Chechik. M, 2013) e a abordagem proposta nesta tese pode também ser alvo de trabalhos futuros.

- *Implementar uma análise de dependência de tarefas da abordagem proposta:* atualmente ao realizar a reconciliação de uma tarefa individualmente, nossa abordagem pode gerar resultados inconsistentes devido a necessidade da aplicação de outras tarefas dependentes desenvolvidas anteriormente. Tais dependências entre tarefas não são atualmente computadas pela nossa abordagem. Dessa forma, um trabalho futuro seria considerar tais dependências.

- *Adicionar outros tipos de conflitos:* a abordagem atualmente identifica diferentes tipos de conflitos de código através da comparação do *source* e do *target*, mas não trata possíveis conflitos gerados após a integração do código,

como os conflitos de alta ordem tratados por Brun et al (Brun. Y et al, 2013), ou os conflitos de linguagem e de testes de Guimarães & Silva (Guimarães, M e Silva, Antônio. 2012). A abordagem pode ser aprimorada para incluir estes tipos de conflitos.

9 Referências Bibliográficas

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M, 1996. *Pattern-Oriented Software Architecture, A System of Patterns*, vol. 1. Wiley,

Apel, S. L., Brandl, J., Lengauer, B. C. & Kästner, C., 2011. *Semistructured Merge: Rethinking Merge in Revision Control Systems*. In Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM Press.

Brun, Y., Holmes, R., Ernst, M. D. & Notkin, D., 2013. *Early Detection of Collaboration Conflicts and Risks*, IEEE Transactions on Software Engineering, vol. 39, no. 10, pp. 1358-1375.

Clements, P. & Northrop, L., 2001. *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional,

CODICE SOFTWARE, 2013. *Semantic Merge - The Ultimate Merging Machine - Intro Guide*, Disponível em http://www.semanticmerge.com/documents/semanticmerge-intro-guide.pdf, Acessado em 05/03/2014

Czarnecki , K. & *Eisenecker, U,* 2000. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

Dewan, P. & Hegde, R., 2007. *Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development.* ECSCW'07: Proceedings of the Tenth European Conference on Computer Supported Cooperative, Limerick, Ireland, Work, 24-28.

Dourish, P. & Bellotti, V, 1992. *Awareness and Coordination in Shared Workspaces*. Proceedings of the ACM Conference on Computer-Supported Cooperative Work,p. 107-114,

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M. & Czarnecki, K., 2013. *An Exploratory Study of Cloning in Industrial Software Product Lines* . 17th European Conference on Software Maintenance and Reengineering (CSMR'13).

Flower, M., 2012. *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Godfrey, M., 2013. *All we like sheep: Copy / paste as principled engineening tool* – Product Line Workshop – 2013 University of Waterloo – Canada.

Greenfield, J. & Short, K., 2005. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools*, John Wiley and Sons.

Guimarães, M. L. & Silva, A. R, 2012. Improving early detection of software merge conflicts. In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 342-352,

Hattari, L. P, 2012. Change-centric Improvement of Team Collaboration, PHD Thesis, Faculty of Informatics of the Università della Svizzera Italiana, Disponível em http://www.inf.usi.ch/faculty/lanza/Downloads/Hatt2012a.pdf

Hattori, L. & Lanza, M., 2010. *Syde: a tool for collaborative software development, Software Engineering*, ACM/IEEE 32nd International Conference on , vol.2, no., pp.235,238, 2-8 May 2010 doi: 10.1145/1810295.1810339.

Herzig, K. S & Zeller, A., 2011. *Mining your own evidence, in Making Software*, pp. 517–529, Sebastopol, Calif., USA: O'Reilly,

J. Bosch, 2002. *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization*, in Software Product Lines. Springer, vol. 2379, pp. 247–262.

Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson, S., 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November,

Kästner, C, 2010. *Virtual Separation of Concerns: Toward Preprocessors* 2.0

Kästner, C; Apel, S; Thüm T; & Saake G, 2011. *Type Checking Annotation-Based Product Lines*. ACM Transactions on Software Engineering and Methodology (TOSEM).

Krueger, C, 2002. *Easing the Transition to Software Mass Customization,* Springer-Verlag London, UK.

Lima, G., 2012. Desafios no Desenvolvimento e Implantação de Grandes Sistemas de Gestão: o caso dos sistemas SIG-UFRN. CBSoft,

Lima, G & Ferreira, A, 2008. Sistemas Institucionais Integrados da UFRN - II Workshop de TI das IFES – Gramado-RS.

Linden, F. van der & Muller, J, 1995. *Creating architectures with building blocks*. IEEE Software, 12(6):51–60.

Linden, F. van der; Schmid, K. & Rommes, E., 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer.

MacKenzie, D., Eggert, P., & Stallman, R., 1997, Comparing and Merging Files with GNU Diff and Patch. Bristol: Network Theory. ISBN 0-9541617-5-0.

Malks, D. Alur, D.; Crupi; J., 2002. *Core J2EE Patterns: Best Practices and Design Strategies*, Sun Microsystem.

Mende, T., Koschke, R. & F. Beckwermert, 2009, An Evaluation of Code Similarity Identification for the Grow-and-Prune Model, J. of Soft. Maintenance and Evolution, vol. 21, no. 2, pp. 143–169.

Mens, T., 2002. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering (TSE), 28(5):449–462.

Neighbors, J, 1984. *The draco approach to constructing software from reusable components*. IEEE Transactions on Software Engineering, 10(5):564–573.

Parnas, D., 1976. *On the design and development of program families*. IEEE Transactions on Software Engineering, 2(1):1–9.

Pereira, D., Lima, G., Kulesza, U., 2010. Padrões de Projeto aplicados ao desenvolvimento da Arquitetura dos Sistemas Institucionais da Universidade Federal do Rio Grande do Norte. Proceedings of 8th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'2010).

Poulin, J, 1997. *Measuring Software Reuse — Principles, Practices, and Economic Models*. Addison-Wesley.

Riva, C & Del Rosso, C., 2003. *Experiences with Software Product Family Evolution*, in Proc. of IWPSE'03 workshop on Principles of Software Evolution, , pp. 161.

Rubin, J. & Chechik M., 2012 , "Combining Related Products into Product Lines," in Proc. of FASE'12, pp. 285–300.

Rubin, J. & Chechik, M., 2013. *A framework for managing cloned product variants*. In*Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1233-1236.

Rubin, J., Czarnecki, K. & Chechik, M, 2013. *Managing cloned variants: a framework and experience.* In Proceedings of the 17th International Software Product Line Conference (SPLC '13). ACM, New York, NY, USA, 101-110, DOI=10.1145/2491627.2491644 http://doi.acm.org/10.1145/2491627.2491644

Rubin, J., Kirshin, A., Botterweck, G. & Chechik, M., 2012. *Managing forked product variants*. In *Proceedings of the 16th International Software Product Line Conference - Volume 1* (SPLC '12), Vol. 1. ACM, New York, NY, USA, 156-160,. DOI=10.1145/2362536.2362558 http://doi.acm.org/10.1145/2362536.2362558

Santos, J., Lima, J., Lima, G. & Kulesza, U, 2012. Execução Condicional: Um Padrão para Implementação de Variabilidades de Granularidade Fina, SugarLoap.

Sarma, A., Noroozi, Z. & Van der Hoek, Andre, 2003. *Palantir: raising awareness among configuration management workspaces*, Software Engineering, 2003. Proceedings. 25th International Conference on , vol., no., pp.444,454, 3-10. doi: 10.1109/ICSE.2003.1201222

Sarma, A., Redmiles, D. F. & van der Hoek, A., 2012. *Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes*, IEEE Transactions on Software Engineering, vol. 38, no. 4, pp. 889-908.

SEI - Software Engineering Institute, 2012. *A Framework for Software Product Line Practice*, Version 5.0, Carnegie Mellon. Acessado em 12/01/2013. Disponível em http://www.sei.cmu.edu/productlines/frame_report/pl_is_not.htm;

Svahnberg, M., van Gurp, J., Bosch, J., 2001, *On the Notion of Variability in Software Product Lines*, In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)

Weiss, D & Lai, C., 1999. *Software Product-Line Engineering: A Family- Based Software Development Process*. Addison-Wesley Professional.

Wloka, J., Ryder, B., Tip, F. & Ren, X., 2009. *Safe-commit analysis to facilitate team software development.* In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 507-517, DOI=10.1109/ICSE.2009.5070549 http://dx.doi.org/10.1109/ICSE.2009.5070549

Zimmermann, T, 2007. Mining Workspace Updates in CVS. In Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07), IEEE Computer Society, Washington, DC, USA, 11-. DOI=10.1109/MSR.2007.22 http://dx.doi.org/10.1109/MSR.2007.22.

10. Apêndice A – Dados Coletados no Estudo Empírico

10.1 Dados complementares da QP1 - Que tipos de conflitos de código ocorrem durante a evolução e *merge* das linhas de produtos de software clonadas?

QP.1.3 – Qual a frequência de conflitos que ocorrem por tarefa no source e no target?

Total de conflitos por Tarefa	**E1**			**E2**			**E3**			**E4**			**Total**		
Tipo de Conflito	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**
0	626	677	656	208	234	208	729	797	685	574	738	583	2137	2446	2132
1	83	39	34	24	3	14	75	10	50	163	52	37	345	104	135
2	11	6	18	5	1	2	3	2	15	42	11	23	61	20	58
3	4	3	4	1	0	2	1	0	4	15	5	17	21	8	27
4	2	1	2	0	0	2	0	0	11	7	0	10	9	1	25
5	0	0	1	0	0	1	1	0	11	3	2	10	4	2	23
10	0	0	10	0	0	4	0	0	12	4	0	47	4	0	73
20	0	0	1	0	0	4	0	0	13	1	0	28	1	0	46
Mais	0	0	0	0	0	1	0	0	8	0	1	54	0	1	63

Tabela 10-1 Histograma por cada evolução paralela - *Source*

Total de conflitos por Tarefa	**E1**			**E2**			**E3**			**E4**			**Total**		
Tipo de Conflito	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**
0	33	34	27	43	54	28	34	46	21	170	187	75	280	321	151
1	2	10	2	5	2	8	5	8	5	15	27	26	27	47	41
2	6	1	4	3	1	8	5	2	0	12	8	16	26	12	28
3	2	2	4	2	0	2	4	1	1	7	4	12	15	7	19
4	2	3	3	1	0	5	0	0	5	3	6	12	6	9	25
5	2	1	3	3	0	1	0	0	3	3	1	5	8	2	12
10	3	1	5	0	0	2	4	0	6	17	4	23	24	5	36
20	1	0	4	0	0	2	5	0	5	8	1	29	14	1	40
Mais	3	2	2	0	0	1	0	0	11	3	0	40	6	2	54

Tabela 10-2 Histograma por cada evolução paralela - Target

QP-1.5 - Qual a distribuição dos conflitos nas diversas operações atômicas?

	E1		**E2**		**E3**		**E4**	
Tipo de Conflito	D	I	D	I	D	I	D	I
Field Add	0	21	0	0	0	1	0	6
Field Delete	0	12	0	0	0	18	2	33
Field Update	0	11	0	4	0	33	4	39
Method Add	3	15	0	0	0	9	0	10
Method Delete	0	29	0	0	0	6	5	18
Method Update	72	422	7	44	13	104	124	220
Total	**75**	**510**	**7**	**48**	**13**	**171**	**135**	**326**

Tabela 10-3 Conflitos por Tipo de Operação Atômica - *Source*

	E1		**E2**		**E3**		**E4**	
Tipo de Conflito	D	I	D	I	D	I	D	I
Field Add	0	0	0	0	0	0	0	0
Field Delete	0	0	0	0	0	0	2	0
Field Update	0	0	0	0	0	0	4	2
Method Add	0	0	0	2	0	5	0	21
Method Delete	2	0	0	0	0	0	0	0
Method Update	40	23	4	21	13	35	83	182
Total	**42**	**23**	**4**	**23**	**13**	**40**	**89**	**205**

Tabela 10-4 Conflitos por Tipo de Operação Atômica – Target

10.2 Dados Complementares da QP2 - Que tipos de tarefas – correção de *bugs*, novas funcionalidades, aprimoramentos – trazem mais conflitos nas linhas de produto de software clonadas?

	E1			E2			E3			E4			Total		
Tipo de Tarefa	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**
Correção de Bugs	391	90	23	169	38	22	389	64	16	389	148	38	1338	340	25,4
Melhoria de Funcionalidades	310	70	23	46	13	28	393	105	27	393	192	49	1142	380	33,3
Novos Casos de Usos	25	4	16	23	1	4	27	4	15	27	14	52	102	23	22,5
	726	164	23	238	52	22	809	173	21	809	354	44	2582	743	28,8

Tabela 10-5 Conflitos por Tipo de Tarefa das evoluções clonadas do *Source*

	E1			E2			E3			E4			Total		
Tipo de Tarefa	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**	**Tarefas**	**Tarefas com Conflitos**	**%**
Correção de Bugs	35	20	57%	47	27	57%	47	30	64	169	114	67%	298	191	64,1
Melhoria de Funcionalidades	15	14	93%	8	5	63%	8	4	50	46	40	87%	77	63	81,8
Novos Casos de Usos	4	3	75%	2	1	50%	2	2	100	23	21	91%	31	27	87,1
	54	37	69%	57	33	58%	57	36	63	238	175	74%	406	281	69,2

Tabela 10-6 Conflitos por Tipo de Tarefa das evoluções clonadas do *Target*

Gráficos dos tipos de conflitos por tipo de tarefa - Source:

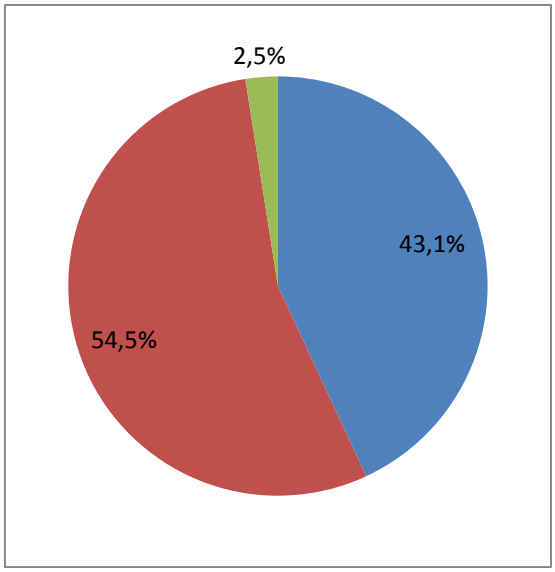


Figura 10-1 Conflitos diretos por Tipo de Tarefa

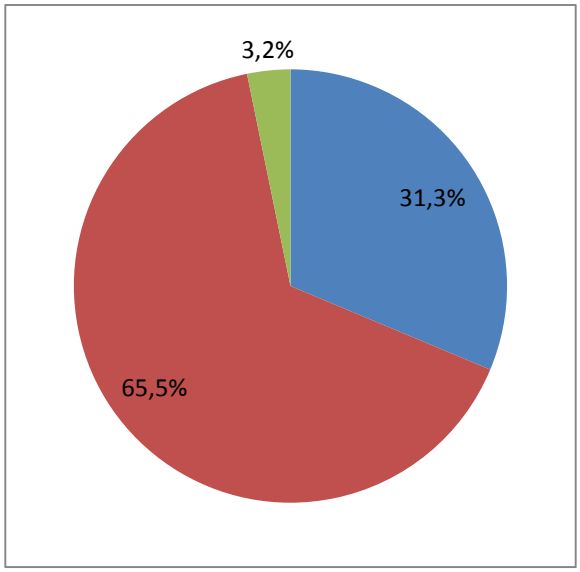


Figura 10-2 Conflitos Indiretos por Tipo de Tarefa

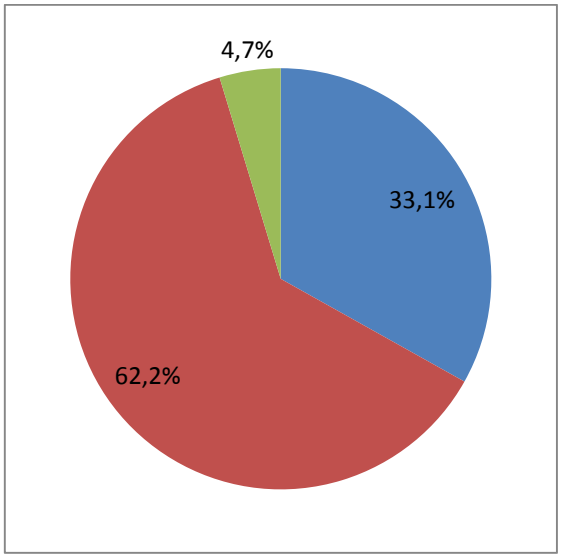


Figura 10-3 Conflitos Textuais por Tipo de Tarefa

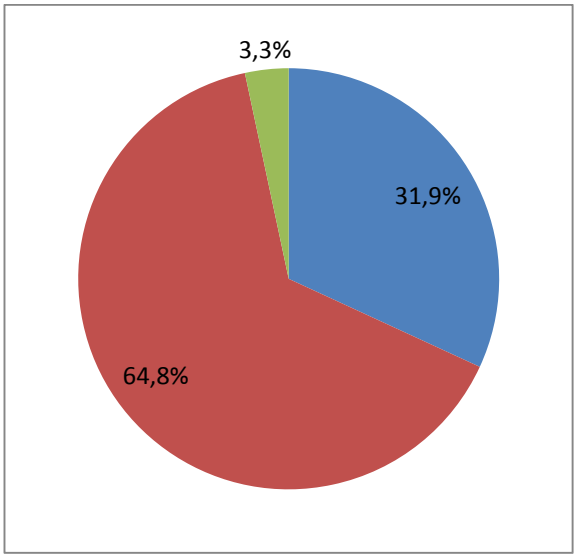
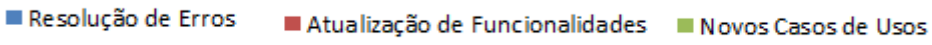


Figura 10-4 Sumário de todos os conflitos por Tipo de Tarefa

Legenda:



Gráficos dos tipos de conflitos por tipo de tarefa - Target:

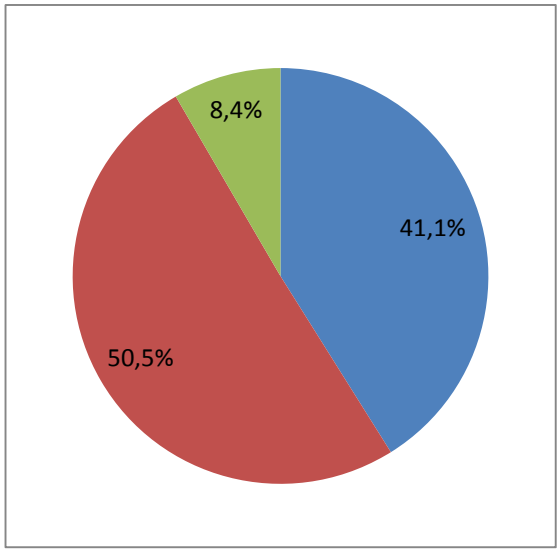


Figura 10-5 Conflitos diretos por Tipo de Tarefa

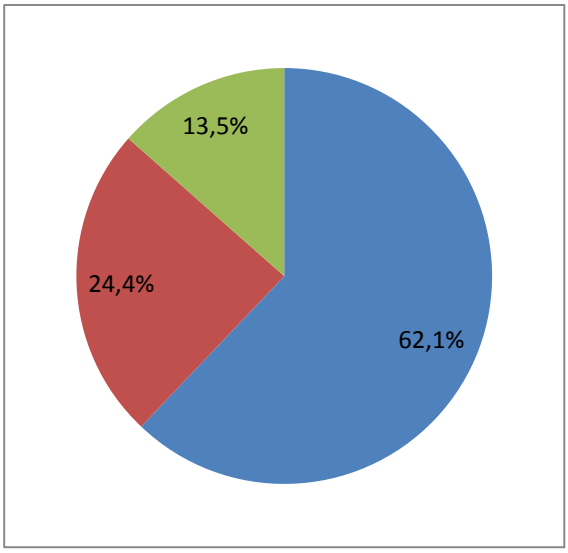


Figura 10-6 Conflitos Indiretos por Tipo de Tarefa

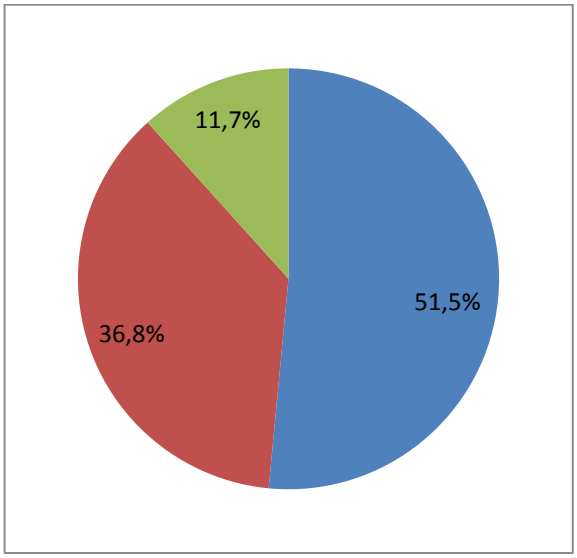


Figura 10-7 Conflitos Textuais por Tipo de Tarefa

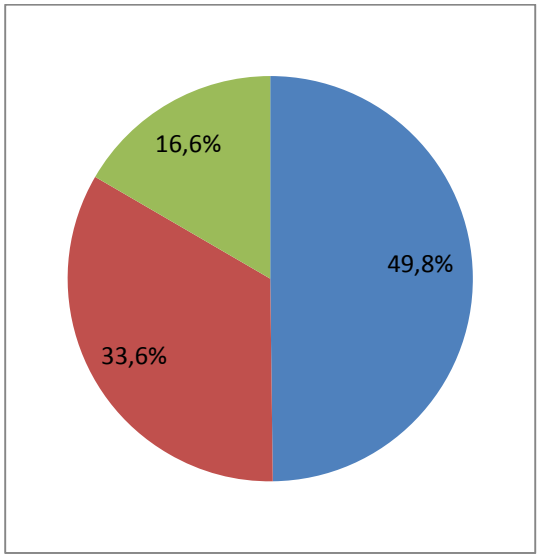
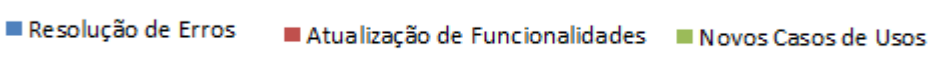


Figura 10-8 Sumário de todos os conflitos por Tipo de Tarefa

Legenda:



10.3 Dados Complementares da QP3 - Onde e como os conflitos ocorrem ao longo das camadas da LPS?

	E1					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	16	23,2%	22	5,4%	14	8,2%
Camada de Acesso a Dados	19	27,5%	14	3,5%	60	35,1%
Camada de Entidade (Domínio)	2	2,9%	17	4,2%	14	8,2%
Outras camadas	10	14,5%	66	16,3%	6	3,5%
Camada Web	17	24,6%	59	14,6%	31	18,1%
Total de Conflitos	64	100%	178	100%	125	100%

Tabela 10-7 Conflitos por Camadas – E1 - *Source*

	E2					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	1	14,3%	20	16,5%	0	0,0%
Camada de Acesso a Dados	2	28,6%	7	5,8%	13	41,9%
Camada de Entidade (Domínio)	0	0,0%	49	40,5%	6	19,4%
Outras camadas	0	0,0%	11	9,1%	1	3,2%
Camada Web	1	14,3%	34	28,1%	11	35,5%
Total de Conflitos	4	100%	121	100,0%	31	100,0%

Tabela 10-8 Conflitos por Camadas - E2 - *Source*

	E3					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	1	7,1%	53	6,1%	2	2,2%
Camada de Acesso a Dados	4	28,6%	27	3,1%	40	44,9%
Camada de Entidade (Domínio)	1	7,1%	516	59,4%	13	14,6%
Outras camadas	0	0,0%	27	3,1%	3	3,4%
Camada Web	8	57,1%	246	28,3%	31	34,8%
Total de Conflitos	14	100,0%	869	100,0%	89	78,8%

Tabela 10-9 Conflitos por Camada - E3 - *Source*

	E4					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	13	10,8%	497	12,0%	93	25,1%
Camada de Acesso a Dados	3	2,5%	210	5,1%	72	19,4%
Camada de Entidade (Domínio)	9	7,5%	1224	29,7%	50	13,5%
Outras camadas	10	8,3%	149	3,6%	25	6,7%
Camada Web	85	70,8%	2046	49,6%	131	35,3%
Total de Conflitos	120	100,0%	4126	100,0%	371	100,0%

Tabela 10-10 Conflitos por Camada - E4 - *Source*

	E1					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	16	25,0%	28	15,7%	14	11,2%
Camada de Acesso a Dados	19	54,3%	31	17,4%	60	48,0%
Camada de Entidade (Domínio)	2	5,7%	26	14,6%	14	11,2%
Outras camadas	10	28,6%	31	17,4%	6	4,8%
Camada Web	17	48,6%	62	34,8%	31	24,8%
Total de Conflitos	64	100%	178	100,0%	125	100,0%

Tabela 10-11 Conflitos por Camada - E1 – *Target*

	E2					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	1	25,0%	39	32,2%	0	0,0%
Camada de Acesso a Dados	2	50,0%	12	9,9%	13	41,9%
Camada de Entidade (Domínio)	0	0,0%	14	11,6%	6	19,4%
Outras camadas	0	0,0%	4	3,3%	1	3,2%
Camada Web	1	25,0%	52	43,0%	11	35,5%
Total de Conflitos	4	100,0%	121	100,0%	31	100,0%

Tabela 10-12 Conflitos por Camada - E2 – *Target*

	E3					
Tipo de Conflito	D		I		T	
Camada de Negócio	1	7,1%	170	19,6%	2	2,2%
Camada de Acesso a Dados	4	28,6%	14	1,6%	40	44,9%
Camada de Entidade (Domínio)	1	7,1%	135	15,6%	13	14,6%
Outras camadas	0	0,0%	9	1,0%	3	3,4%
Camada Web	8	57,1%	540	62,2%	31	34,8%
Total de Conflitos	14	100,0%	868	100,0%	89	100,0%

Tabela 10-13 Conflitos por Camada - E3 – *Target*

	E4					
Tipo de Conflito	**D**		**I**		**T**	
Camada de Negócio	13	10,8%	383	9,4%	93	26,1%
Camada de Acesso a Dados	3	2,5%	181	4,4%	72	20,2%
Camada de Entidade (Domínio)	9	7,5%	1655	40,6%	50	14,0%
Outras camadas	10	8,3%	75	1,8%	25	7,0%
Camada Web	85	70,8%	1782	43,7%	117	32,8%
Total de Conflitos	120	100,0%	4076	100,0%	357	100,0%

Tabela 10-14 Conflitos por Camada - E4 – *Target*

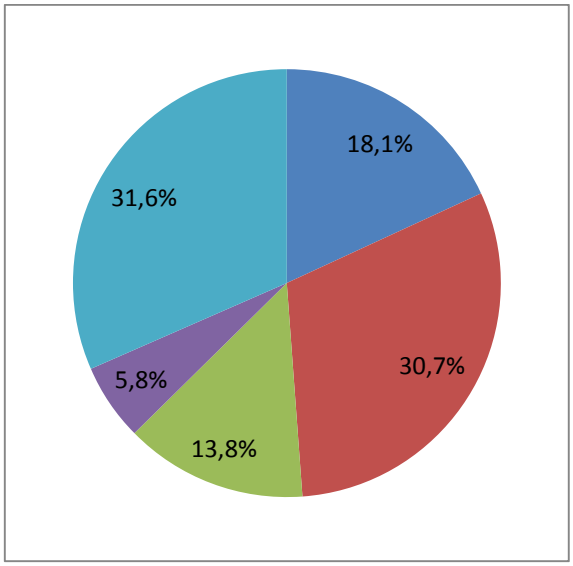


Figura 10-9 Conflitos Textuais por Camada – Target

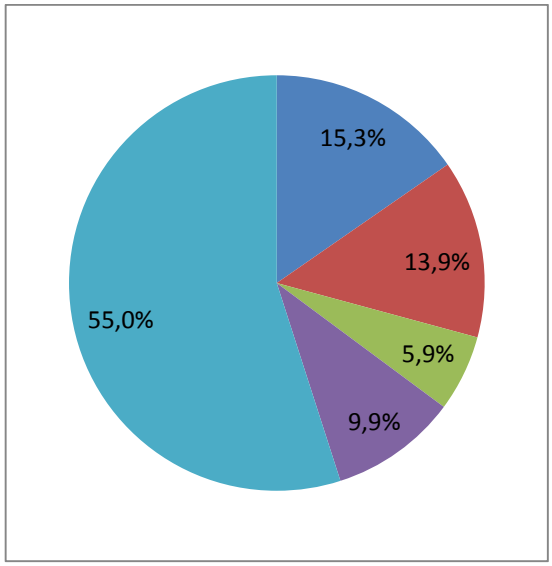


Figura 10-10 Conflitos Diretos por Camada - Target

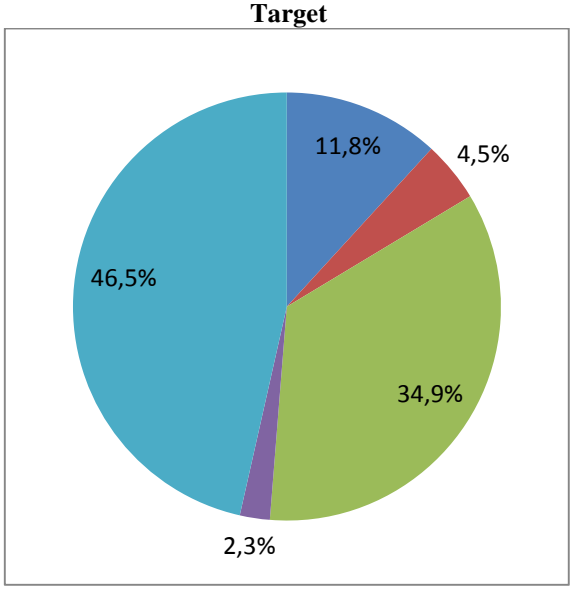


Figura 10-11 Conflitos Indiretos por Camada - Target

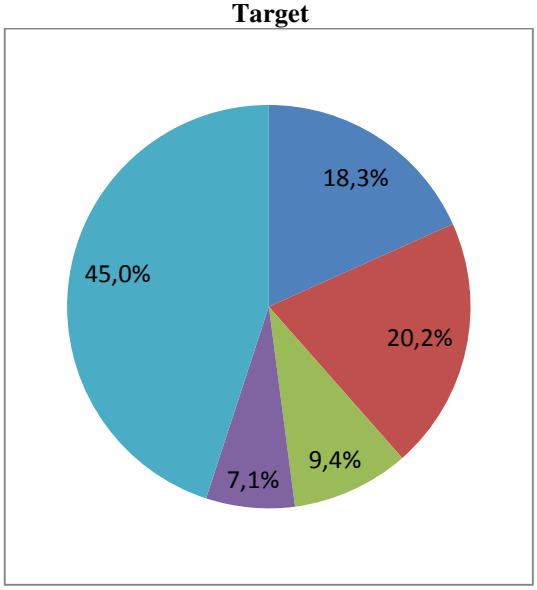
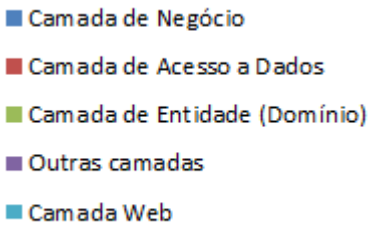


Figura 10-12 Conflitos por Camada do *Target* - Síntese

Legenda:



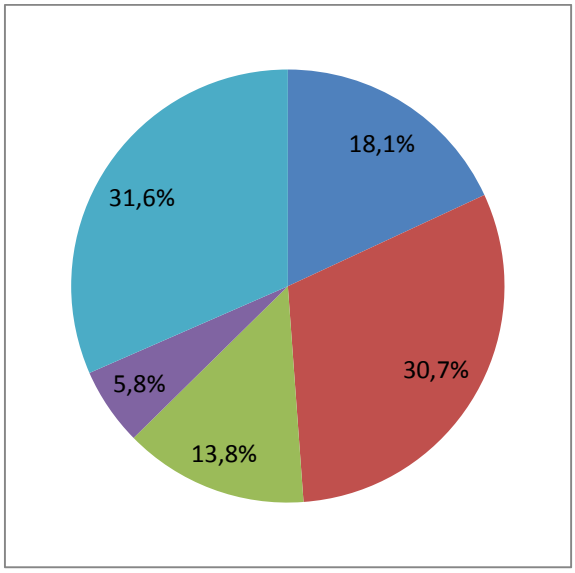


Figura 10-13 Conflitos Textuais por Camada - *Source*

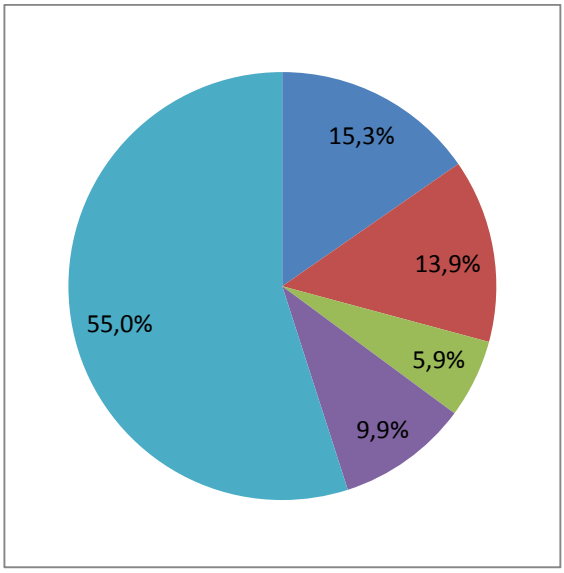


Figura 10-14 Conflitos Diretos por Camada - *Source*

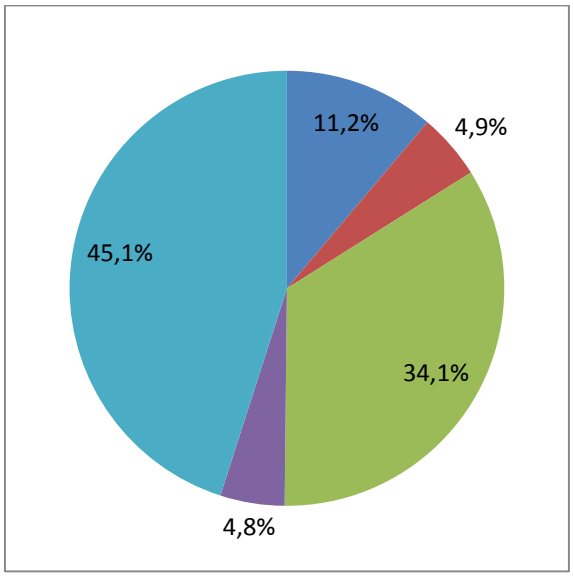


Figura 10-15 Conflitos Indiretos por Camada - *Source*

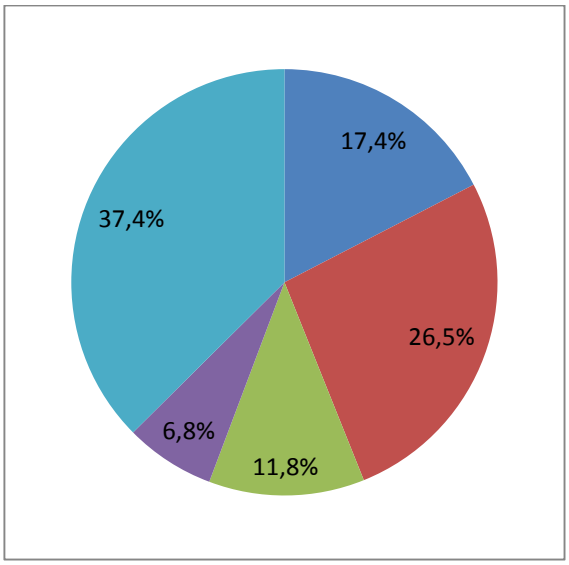
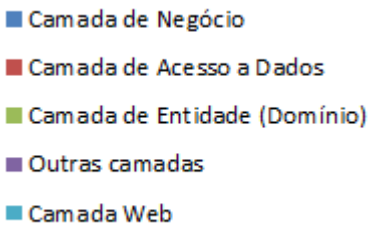


Figura 10-16 Conflitos por Camada do *Source* - Síntese

Legenda:



Distribuições dos conflitos por combinações de camadas no Source e no Target:

Camada	**E1**			**E2**			**E3**			**E4**			**Total**		
Tipo de Conflito	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**
Negócio	9	4	6	1	4	0	0	8	1	3	10	10	13	26	17
Acesso a Dados	11	6	21	0	0	6	1	5	15	1	12	13	13	23	55
Entidade	0	4	2	0	0	0	0	4	1	0	6	5	0	14	8
Outras	3	9	1	0	2	1	0	3	1	1	5	1	4	19	4
Web	7	7	11	2	4	6	4	21	17	23	47	46	36	79	80
Negócio e Acesso a Dados	0	0	2	0	1	0	0	3	2	0	3	2	0	7	6
Negócio e Entidade	0	0	0	0	0	0	0	0	1	0	3	3	0	3	4
Negócio e Outros	2	2	1	0	0	0	0	1	1	0	1	1	2	4	3
Acesso a Dados e Entidade	0	1	0	0	0	0	1	0	3	1	4	5	2	5	8
Acesso a Dados e Outras	3	3	5	0	0	0	0	0	1	0	1	1	3	4	7
Entidade e Outras	0	2	0	0	0	0	0	0	0	0	0	0	0	2	0
Web e Negócio	0	0	0	0	2	0	1	1	1	5	9	12	6	12	13
Web e Acesso a Dados	1	3	8	0	2	1	0	6	4	3	12	10	4	23	23
Web e Entidade	0	2	1	0	2	2	1	7	3	5	13	17	6	24	23
Web e Outras	1	2	3	0	1	0	0	1	2	2	4	2	3	8	7
Negócio, Acesso a Dados e Entidade	0	0	0	0	0	0	0	2	0	1	3	2	1	5	2
Negócio, Acesso a Dados e Outras	0	1	2	0	0	0	1	1	1	0	1	1	1	3	4
Negócio, Entidade e Outras	1	1	1	0	0	0	0	1	0	0	1	0	1	3	1
Acesso a Dados, Entidade e Outras	0	0	2	0	0	0	0	2	1	0	2	1	0	4	4
Web, Negócio e Acesso a Dados	1	1	3	0	0	0	0	5	3	1	9	9	2	15	15
Web, Negócio e Entidade	0	1	0	0	3	3	0	6	1	3	10	13	3	20	17
Web, Negócio e Outras	1	2	2	0	0	0	0	0	0	0	2	2	1	4	4
Web, Acesso a Dados e Outras	0	0	2	0	2	3	0	7	3	2	9	13	2	18	21

Web, Acesso a Dados e Outras	0	0	1	0	1	1	0	1	0	1	2	3	1	4	5
Web, Entidade e Outras	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
Negócio, Acesso a Dados e Outras	1	2	1	0	0	0	0	2	1	1	2	2	2	6	4
Web, Negócio, Acesso a Dados e Entidade	2	2	10	0	2	3	3	17	11	9	31	39	14	52	63
Web, Negócio, Acesso a dados e Outros	0	0	1	0	0	0	0	2	1	1	2	3	1	4	5
Web, Negócio, Entidade e Outros	1	4	2	0	0	0	0	1	0	0	3	0	1	8	2
Web, Acesso a Dados e Outros	1	2	2	0	0	1	0	3	0	1	1	1	2	6	4
Conflitos em Apenas Uma Camada	30	30	41	3	10	13	5	41	35	28	80	75	66	161	164
Conflitos em Duas Camadas	7	15	12	0	8	3	3	19	10	16	50	41	26	92	66
Conflitos em Três Camadas	3	7	5	0	6	7	1	25	4	8	39	31	12	77	47
Conflitos em Quatro Camadas	5	13	16	0	5	4	3	33	13	12	52	45	20	103	78
Conflitos em Todas as Camadas	4	8	10	1	2	1	0	14	5	7	18	18	12	42	34
Total com Conflitos	**49**	**73**	**84**	**4**	**31**	**28**	**12**	**132**	**67**	**71**	**239**	**210**	**136**	**475**	**389**

Tabela 10-15 Combinação de conflitos por camadas - *Source*

Camada	**E1**			**E2**			**E3**			**E4**			**Total**		
Tipo de Conflito	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**	**D**	**I**	**T**
Negócio	5	2	1	0	6	0	0	4	1	2	14	7	7	26	9
Acesso a Dados	1	3	3	1	0	2	0	3	3	1	11	3	3	17	11
Entidade	0	0	0	0	1	0	0	1	0	0	3	0	0	5	0
Outras	1	3	2	0	0	0	0	0	0	0	2	2	1	5	4
Web	3	1	3	1	9	3	7	11	8	14	50	17	25	71	31
Negócio e Acesso a Dados	1	0	0	1	2	0	1	2	2	0	2	0	3	6	2
Negócio e Entidade	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
Negócio e Outros	0	0	0	0	1	1	0	1	1	0	0	0	0	2	2
Acesso a Dados e Entidade	0	0	0	0	1	1	1	1	1	0	2	0	1	4	2
Acesso a Dados e Outras	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
Entidade e Outras	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Web e Negócio	2	4	1	0	2	0	0	2	0	1	6	1	3	14	2
Web e Acesso a Dados	1	2	2	0	1	0	0	2	0	4	12	2	5	17	4
Web e Entidade	0	0	0	0	1	1	0	2	0	5	8	4	5	11	5
Web e Outras	0	0	0	0	0	0	0	0	0	2	1	0	2	1	0
Negócio, Acesso a Dados e Entidade	0	0	0	0	1	1	1	1	1	0	0	0	1	2	2
Negócio, Acesso a Dados e Outras	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Negócio, Entidade e Outras	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Acesso a Dados, Entidade e Outras	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
Web, Negócio e Acesso a Dados	0	2	1	0	0	0	0	0	0	1	8	4	1	10	5
Web, Negócio e Entidade	0	1	0	0	0	0	0	0	0	3	7	6	3	8	6
Web, Negócio e Outras	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Web, Acesso a Dados e Outras	1	1	1	0	0	0	0	1	1	2	13	5	3	15	7
Web, Acesso a Dados e Outras	0	0	0	0	0	0	0	0	0	2	3	2	2	3	2
Web, Entidade e Outras	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1

Negócio, Acesso a Dados e Outras	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1
Web, Negócio, Acesso a Dados e Entidade	1	3	2	0	2	2	0	3	2	3	8	5	4	16	11
Web, Negócio, Acesso a dados e Outros	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Web, Negócio, Entidade e Outros	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Web, Acesso a Dados e Outros	0	0	0	0	0	0	0	0	0	3	4	2	3	4	2
Conflitos em Apenas Uma Camada	10	9	9	2	16	5	7	19	12	17	80	29	36	124	55
Conflitos em Duas Camadas	5	7	4	1	8	3	2	10	0	12	32	7	20	57	11
Conflitos em Três Camadas	2	5	3	0	1	1	1	2	1	8	32	14	11	40	16
Conflitos em Quatro Camadas	1	4	3	0	2	2	0	4	2	6	29	7	7	40	14
Conflitos em Todas as Camadas	1	2	1	0	0	0	0	0	0	3	4	2	4	6	3
Total com Conflitos	**19**	**27**	**20**	**3**	**27**	**11**	**10**	**35**	**15**	**46**	**177**	**59**	**78**	**267**	**99**

Tabela 10-16 Combinação de camadas nos conflitos - Target