UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA MODULARIZAÇÃO DE FRAMEWORKS DE MÚLTIPLOS DOMÍNIOS EM LINHA DE PRODUTOS DE FRAMEWORKS

VICTOR HUGO SANTIAGO COSTA PINTO

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP
Agosto/2013

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA MODULARIZAÇÃO DE FRAMEWORKS DE MÚLTIPLOS DOMÍNIOS EM LINHA DE PRODUTOS DE FRAMEWORKS

VICTOR HUGO SANTIAGO COSTA PINTO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientador: Dr. Valter Vieira de Camargo

São Carlos - SP
Agosto/2013

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P659am

Pinto, Victor Hugo Santiago Costa.
Uma abordagem para modularização de frameworks de múltiplos domínios em linha de produtos de frameworks / Victor Hugo Santiago Costa Pinto. -- São Carlos : UFSCar, 2014.
170 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2013.

1. Engenharia de software. 2. Reuso. 3. Frameworks. 4. Linha de produtos de frameworks. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Uma Abordagem para Modularização de Frameworks de Múltiplos Domínios em Linha de Produtos de Frameworks"

Victor Hugo Santiago Costa Pinto

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Valter Vieira de Camargo
(Orientador - DC/UFSCar)

Rosana Braga

Profa. Dra. Rosana Teresinha Vaccare Braga
(ICMC/USP)

Prof. Dr. Marco Tulio de Oliveira Valente
(UFMG)

São Carlos
Outubro/2013

Deixe suas esperanças, e não seus ferimentos, moldarem seu futuro.

(Robert H. Schuller)

AGRADECIMENTO

Foram muitos os que, direta ou indiretamente, me ajudaram a concluir este trabalho. Minha sincera gratidão...

... a Deus que me abençoou abundantemente, abrindo portas intransponíveis, permitindo que eu chegasse até aqui. Sem o auxílio Dele nada disso seria possível.

...à minha família, que apesar da distância, sempre estiveram presentes em pensamentos e orações. Em especial a minha mãe Cineide que sempre lutou para oferecer a mim e as minhas irmãs um futuro melhor. Seu exemplo de vida me orgulha e motiva para continuar lutando.

...a minha noiva Paula também merece grande parte dessa conquista, pois o término de um trabalho como este sem dúvida não seria possível sem o apoio emocional que ela me concede.

...ao meu orientador Valter Camargo, pela oportunidade, confiança, palavras de incentivo, também por toda ajuda e ensinamentos ao longo desses anos.

...aos meus amigos mais próximos com quem sempre pude contar: Daniel San Martín, Rafael Durelli e Matheus Viana.

...a todos os colegas do AdvanSE e do PPG-CC, pelo companheirismo, receptividade e eventuais apoios.

... aos professores, pelos ensinamentos, e à secretaria e funcionários do DC, pelo apoio.

... aos alunos da turma de 2012 da disciplina Tópicos em Engenharia de Software da UFSCar, pelo interesse e participação no experimento realizado neste trabalho

... à CAPES, ao CNPq e ao PPG-CC, pelo apoio financeiro.

Enfim... a todos que, de alguma maneira, contribuíram para realização deste sonho e torceram para que tudo desse certo.

Que Deus abençoe todos vocês!

Muito obrigado!

RESUMO

Frameworks são ferramentas de reúso que contribuem para a redução de custos e aumento da produtividade no desenvolvimento de aplicações. Eles são amplamente empregados atualmente e tendem a fornecer um conjunto satisfatório de variabilidades de um determinado domínio. Em geral, uma característica comum no processo de evolução de frameworks é a adição de novas variabilidades na tentativa de atender demandas de um conjunto cada vez maior de usuários. Entretanto, quando tais evoluções não são bem planejadas e gerenciadas, a arquitetura original do framework acaba se desviando da que havia sido previamente planejada, resultando em uma arquitetura complexa e inflexível. Além disso, as novas variabilidades adicionadas podem pertencer a domínios que não foram originalmente planejadas para o framework; levando ao que chamamos de Frameworks de Múltiplos Domínios (FMDs). Um problema desse tipo de framework é que determinadas variabilidades podem ser inúteis para certas aplicações apoiadas pelo framework. Dessa forma, FMDs apresentam problemas tanto para os engenheiros de aplicação (EA) quanto para os engenheiros do framework (EF). No primeiro caso, a curva de aprendizado e a produtividade são comprometidas, pois o EA precisará conviver com um conjunto grande de variabilidades que podem ser desnecessárias. Para os EFs, a inflexibilidade arquitetural acaba dificultando manutenções e a composição/decomposições de versões menores e mais restritas do framework. Nesse contexto, como uma alternativa para os problemas supracitados, apresenta-se neste trabalho uma abordagem para reestruturar FMDs em Linhas de Produtos de Frameworks (LPFs). Uma LPF é uma linha de produtos em que os membros gerados são frameworks ao invés de aplicações prontas. A ideia principal é que a flexibilidade dessa nova arquitetura permita a geração de frameworks menores e mais direcionados aos requisitos de um domínio/subdomínio, evitando a presença de features/variabilidades que nunca serão usadas. Um dos pontos chave da abordagem proposta consiste em determinar o cenário de utilização que a LPF deve satisfazer. Assim, pode-se decompor um FMD em features com níveis adequados de granularidade, fator que impacta diretamente na qualidade da LPF. Para a concepção da abordagem conduziu-se um estudo de caso no qual o framework de aplicação GRENJ foi transformado em uma LPF. Foram realizados dois tipos de avaliação. O primeiro foi um experimento para comparar o esforço de remodularizar um FMD em LPF usando programação orientada a aspectos e desenvolvimento dirigido a modelos. O segundo foi um estudo que comparou aplicações desenvolvidas com o apoio do framework original com aplicações desenvolvidas a partir da LPF resultante. Os resultados apontaram vantagens com relação à redução dos esforços e aumento da produtividade.

Palavras-chave: Reuso de software, Frameworks, Linha de Produtos de Frameworks.

ABSTRACT

Frameworks are tools for software reuse that contribute for reducing costs and increased productivity in application development. Nowadays they are widely used and they tend to provide a satisfactory set of variabilities of a given domain. In general, a common trend in the evolution of these frameworks is the addition of new variabilities in attempting to address demands of a growing set of users. However, when such evolutions are not well designed and managed, the original architecture of the framework ends deviating from what had been previously planned, resulting in a complex and inflexible architecture. In addition, the new variabilities may belong to domains that were not originally planned for the framework, and become what we call Multiple Domains Frameworks (MDF). A problem of this kind of framework is that some variabilities may be useless for certain applications supported by the framework. Thus, MDF have problems for the Application Engineers (AE) and for Framework Engineers (FE). In the first case, the learning curve and the productivity are compromised, because AE will need to live together with a vast set of variabilities that may be unnecessary. For FE, the inflexibility of architecture complicates maintenance and composition/decomposition of smaller and more restrict versions of the framework. In this context, as an alternative to the aforementioned problems, we present an approach for modularization of MDFs into Framework Product Lines (FPL). An FPL is a product line in which the generated members are frameworks instead of applications. The main idea is that flexibility of this new architecture allows the generation of smaller and directed frameworks to the requirements of a domain/subdomain, avoiding the presence of features/variabilities that will never be used. One of the key points of this approach is to determine the Usage Scenario that the FPL must satisfy. Thus, an MDF can be decomposed in features with appropriate levels of granularity, a factor that directly impacts in the quality of an FPL. For the design of this approach, we conducted a case study in which the application framework GRENJ was modularized into an FPL. Two kinds of evaluation were performed. The first was an experiment to compare the effort to modularize a FMD in FPL using Aspect-Oriented Programming and Model-Driven Development. The second was a comparative study among applications developed with support of original framework and applications developed from the resulting FPL. The results show advantages in the reduction of efforts and increased productivity.

Keywords: Software reuse, Frameworks, Framework Product Lines.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

DSOA – *Desenvolvimento de Software Orientado a Aspectos*

EA – *Engenheiro de Aplicação*

EF – *Engenheiro de Framework*

FMD – *Frameworks de Múltiplos Domínios*

FODA – *Feature-Oriented Domain Analysis*

FOO – *Frameworks Orientados a Objetos*

FOSD - *Feature-Oriented Software Development*

LPF – *Linha de Produtos de Framework*

LPS – *Linha de Produtos de Software*

M2T- *Model to Text*

MDA – *Arquitetura Dirigida a Modelos (Model Driven Architecture)*

MDD – *Desenvolvimento Dirigido a Modelos (Model Driven Development)*

OMG - *Object Management Group*

ORB - *Object Request Broker*

POA – *Programação Orientada a Aspectos*

RMI – *Remote Method Invocation*

SECOS – *Ecossistemas de Software (Software Ecosystems)*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto e Motivação

A reutilização é um conceito antigo da engenharia de software que visa reaproveitar parte do software desenvolvido, reduzindo o custo e aumentando a produtividade no desenvolvimento de novos projetos. A reutilização compreende desde trechos de código a bibliotecas, componentes, serviços web, padrões, frameworks, linhas de produtos e aplicações completas (Sommerville, 2011).

Uma técnica bastante conhecida de reúso são os frameworks. De acordo com Fayad et al. (1999a), frameworks orientados a objetos são soluções de software que visam apoiar o desenvolvimento de aplicações em domínios específicos. O objetivo principal é a reutilização de projeto e código em um esforço para reduzir o tempo de desenvolvimento e aumentar a qualidade das aplicações desenvolvidas. Os frameworks também podem ser definidos como um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação (Fayad e Schmidt, 1997). Os frameworks podem ser utilizados instanciando suas classes, isto é, por meio de herança ou combinando as diversas classes concretas existentes no framework para obter a aplicação desejada.

Dentre os diversos tipos de frameworks, encontram-se os chamados frameworks de aplicação (Fayad et al., 1999a; Fayad e Johnson, 2000) que são voltados para domínios verticais (Rogers, 1997; Fayad et al., 1999b) e os frameworks de infraestrutura, que apoiam domínios horizontais (Gamma et al., 1995; Fayad et al., 1999b). Os frameworks de aplicação são desenvolvidos com base na

experiência obtida em um determinado domínio específico, como gestão de recursos de negócio, clínicas médicas, aplicações robóticas de segurança, etc.

Os frameworks de infraestrutura apoiam o desenvolvimento de partes de uma aplicação, como persistência, segurança, distribuição, interfaces ricas e serviços web, e não de uma aplicação completa. Esses frameworks são facilmente encontrados na web e são também empregados em contexto industrial. O Hibernate[1] é um exemplo desse tipo de framework, que apoia o mapeamento objeto-relacional, de forma que o desenvolvedor não precisa se preocupar com a implementação de toda a parte que controla a persistência.

A maneira como os frameworks são utilizados pode variar dependendo de seu tipo: caixa-branca, caixa-preta ou caixa-cinza (Schmid, 1997; Yassin e Fayad, 2000). Entretanto, independentemente do tipo, geralmente o framework inteiro permanece com a aplicação na versão final, isto é, o código-objeto final do desenvolvimento inclui tanto o código da aplicação quanto o código do(s) framework(s) utilizado(s). Isso significa que não importa o número de variabilidades que estão sendo usadas por uma determinada aplicação; todas elas estarão presentes na versão final. Isso não é visto como um problema, já que a aplicação pode evoluir e as variabilidades remanescentes podem ser necessárias no futuro.

Dessa forma, frameworks começam a ser amplamente aceitos e utilizados quando fornecem um conjunto substancial de variabilidades, de forma que satisfaça grande parte de seus potenciais usuários. Assim, é natural que eles evoluam nessa direção, sempre acrescentando novas variabilidades e funcionalidades. Entretanto, quando essa evolução não é bem planejada e gerenciada, é possível que variabilidades/funcionalidades que não pertencem ao domínio originalmente concebido para o framework, sejam adicionais indiscriminadamente. Quando isso ocorre, tem-se o que chamamos de Frameworks de Múltiplos Domínios (FMDs), já que resulta em um framework que oferece apoio ao desenvolvimento de aplicações em domínios distintos.

O principal problema dos FMDs está relacionado com conjuntos de variabilidades que podem nunca ser utilizadas por determinados conjuntos de aplicações, mas que permanecerão na versão final (Batory et al., 2000; Batory et al., 2011). Como ele apoia o desenvolvimento de aplicações em domínios distintos,

[1] Hibernate. URL: http://www.hibernate.org/

variabilidades de um domínio podem ser inúteis em outro domínio. Assim, engenheiros de aplicação (EA) que utilizam o FMD para desenvolver aplicações para um domínio específico, precisam conviver com um conjunto de variabilidades que nunca serão utilizadas pelas aplicações. Isso pode consumir maior tempo para selecionar somente as apropriadas ou até mesmo, permitir a seleção acidental, prejudicando a produtividade. Além disso, outra característica desses frameworks é que em geral são blocos indivisíveis com arquitetura inflexível e complexa, assim, não é possível remover as variabilidades remanescentes. Essa rigidez impede que engenheiros de frameworks (EF) possam atender demandas por frameworks menores e mais restritos em termos de variabilidades.

1.2 Solução apresentada

Diante dessa problemática, apresenta-se uma solução centrada em dois pontos principais: i) apresentação e refinamento do conceito de “Linha de Produtos de Frameworks (LPFs)” e ii) criação de uma abordagem para transformar FMDs em LPFs.

Uma LPF é uma linha de produtos cujos membros são frameworks, ao invés de aplicações concretas. Isso significa que a composição das features de uma LPF não gera uma aplicação e sim um framework menor, restrito e direcionado a um subdomínio e que ainda precisa ser instanciado ou composto a um código-base para que possa operar (Oliveira et al., 2012). Uma LPF possui uma arquitetura bem mais flexível e divisível, permitindo que frameworks com conjuntos limitados de variabilidades possam ser criados. Embora a ideia geral de uma LPF tenha sido proposta no trabalho de (Oliveira et al., 2012), investigou-se com mais profundidade e observou-se que suas alternativas de implementação precisavam ser mais delineadas.

A abordagem proposta consiste em um conjunto de diretrizes que dá suporte ao EF modularizar um FMD em uma LPF considerando os refinamentos apontados nesta dissertação. A abordagem é independente de linguagem/plataforma específica e deve ser aplicada manualmente, pois não há apoio automático.

Para o desenvolvimento dessa abordagem, foi conduzido um estudo de caso, no qual o framework de aplicação GRENJ (Durelli et al., 2010) foi modularizado segundo os refinamentos do conceito de LPF, obtendo assim uma LPF chamada GRENJ-LPF. Essa LPF permite a composição de membros segundo o domínio e/ou requisitos das aplicações. Assim como no framework original, a instanciação desses membros é totalmente baseada na concretização de métodos e extensão de classes.

1.3 Objetivos e Contribuições

Os objetivos principais da pesquisa são:

- Aumentar a flexibilidade em termos de composição/decomposição das variabilidades dos frameworks, permitindo a geração de frameworks mais direcionados, cobrindo apenas um domínio;
- Melhorar a produtividade dos EAs, evitando a seleção de variabilidades a partir de um conjunto amplo que contém variabilidades susceptíveis de não serem utilizadas.

Como objetivos secundários têm-se:

- Divulgação de um processo de transformação de FMDs em LPFs;
- Disponibilização de uma LPF chamada GRENJ-LPF;
- Divulgação de um plug-in para geração de membros utilizando FeatureIDE[2] e Acceleo[3].

1.4 Resultados

Para a obtenção de resultados foram realizadas duas avaliações, um experimento e um estudo comparativo. No experimento, comparou-se o esforço dos

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

[3] http://www.eclipse.org/acceleo/

participantes para decompor os subdomínios de um FMD em nível de features, constituindo LPFs, com Programação Orientada a Aspectos e por meio de Templates. No estudo comparativo, foram desenvolvidas aplicações utilizando membros de duas LPFs e o FMD precursor. Com base nesses frameworks e aplicações, aplicaram-se métricas para verificar a quantidade de código desnecessário às aplicações, quantidade de variabilidades do framework que não pertenciam ao domínio das aplicações, etc. De modo geral, os resultados foram favoráveis às expectativas iniciais dessas avaliações que incluem redução dos esforços e aumento da produtividade.

1.5 Escopo do Projeto

Este projeto de mestrado faz parte de um projeto maior intitulado: “Infraestrutura de Apoio ao Reúso e Gerenciamento de Famílias de Frameworks Transversais”, que visa agregar habilidades e competências constituindo uma infraestrutura integrada para desenvolvimento de software contendo ferramentas e técnicas de apoio à reutilização e gerenciamento de Famílias de Frameworks Transversais. Este projeto conta com o apoio do CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

1.6 Organização do Trabalho

Esta dissertação está organizada em outros 8 capítulos, além deste de introdução do trabalho. A Figura 1 ilustra a organização geral dos capítulos, além de uma descrição de seus conteúdos.

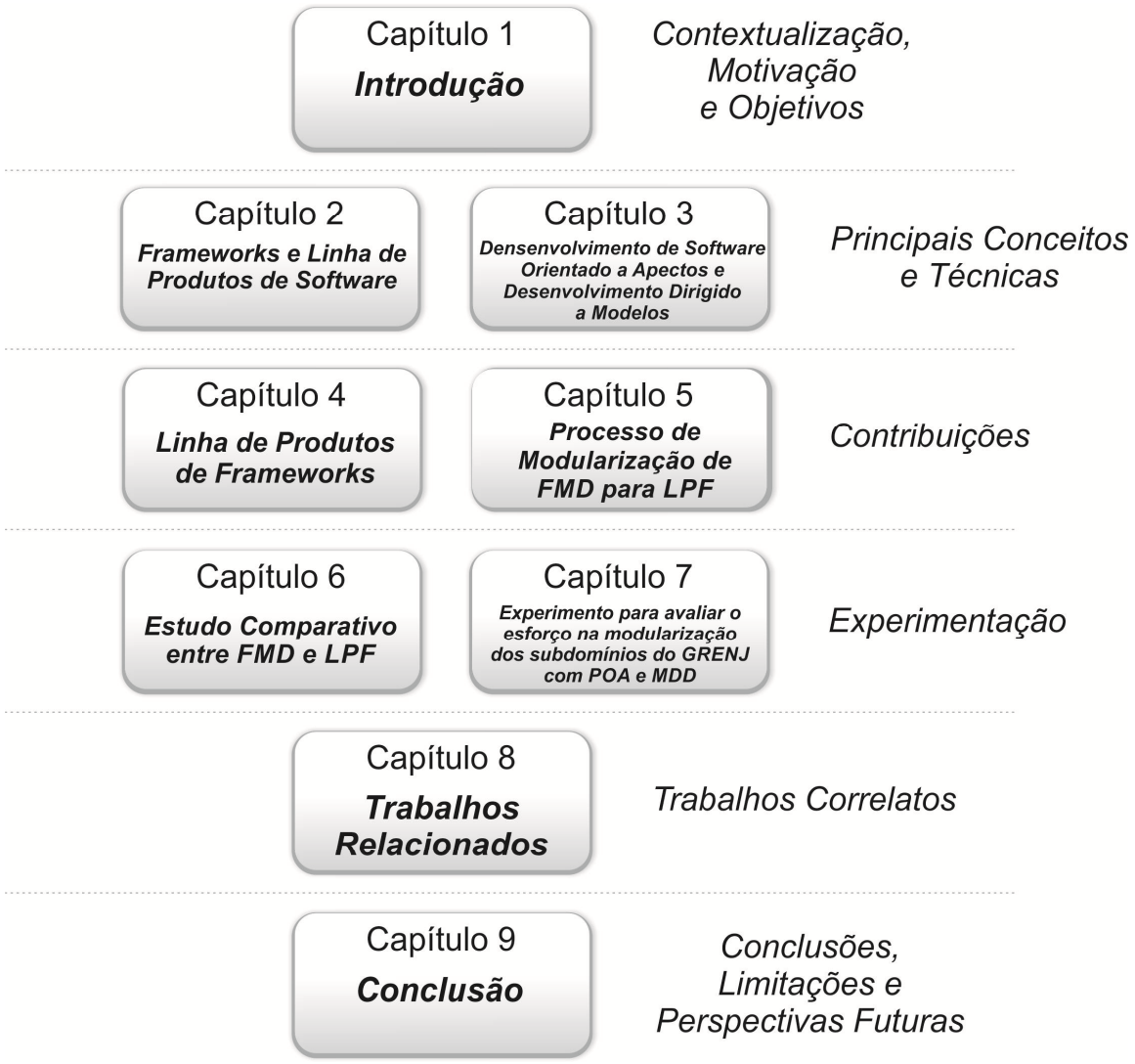


Figura 1: Organização da dissertação

- No **Capítulo 2** é apresenta uma revisão da literatura sobre Reúso de Software, as principais abordagens que apoiam essa prática, destacando os conceitos de Frameworks e Linha de Produtos de Software empregados amplamente no trabalho.
- No **Capítulo 3** são apresentados os conceitos acerca do Desenvolvimento Orientado a Aspectos, Desenvolvimento Dirigido a Modelos, discutindo os principais conceitos envolvidos no desenvolvimento deste trabalho.
- No **Capítulo 4** são abordadas as principais características de Frameworks de Múltiplos Domínios e os refinamentos dos conceitos e particularidades das Linhas de Produtos de Frameworks, destacando as principais diferenças com relação à Linha de Produtos de Software.

- No **Capítulo 5** é apresentado um processo de modularização de Frameworks de Múltiplos Domínios para LPFs. As ideias mais importantes em torno desse processo são discutidas e suas atividades são detalhadas.
- No **Capítulo 6** é apresentado um estudo comparativo entre um Framework de Múltiplos Domínios e membros de duas LPFs. Nesse estudo são discutidas as melhorias obtidas com a modularização e com o apoio de métricas são obtidas informações relevantes que evidenciam tais melhorias.
- No **Capítulo 7** são apresentados os detalhes e os resultados de um experimento em que comparamos o impacto da aplicação das técnicas: Programação Orientada a Aspectos (POA) e Desenvolvimento Dirigido a Modelos – *Model Driven Development* (MDD), na modularização dos subdomínios de um Framework de Múltiplos Domínios no sentido de obter LPFs.
- No **Capítulo 8** são discutidos alguns trabalhos relacionados com a modularização de frameworks e de Linhas de Produtos de Software.
- No **Capítulo 9** são apresentadas as contribuições do trabalho, uma síntese dos principais resultados, às limitações do trabalho e por fim, uma perspectiva quanto aos trabalhos futuros.

Capítulo 2

FRAMEWORKS E LINHA DE PRODUTOS DE SOFTWARE

Neste capítulo apresenta-se uma visão geral sobre reúso de software e as principais abordagens que apoiam essa prática. Dentre essas abordagens, destacamos os Frameworks e as chamadas Linhas de Produtos de Software por serem diretamente relacionados com o trabalho. Assim, seus conceitos e particularidades são explorados neste capítulo. Por fim, na seção de Considerações Finais apresenta-se uma síntese dos pontos discutidos neste capítulo.

2.1 Reúso de Software

A busca por soluções na área de desenvolvimento de software para atender à demanda de software em prazos cada vez mais exigentes, tem impulsionado a criação de práticas que tornem o processo de desenvolvimento mais produtivo, padronizado e consistente.

Dentre essas práticas, tem-se o reúso de software, uma estratégia que visa reduzir custo de produção e manutenção, entrega mais rápida de sistemas e aumento da qualidade de software (Sommerville, 2011). Com a reutilização de software, sistemas previamente desenvolvidos são aproveitados e adaptados para outros contextos, favorecendo a produtividade na criação de novas aplicações (Krueger, 1992).

Para Sommerville, sob o ponto de vista dimensional das unidades de software que são reusáveis, o reúso de software pode ser classificado em três grupos: (1)

sistemas de aplicações (2) componentes e (3) funções. O primeiro grupo compreende a reutilização de aplicações completas, seja modificando alguns módulos ou incorporando uma aplicação sem alterações à outra. As famílias de aplicações que possuem uma arquitetura comum e podem ser adaptadas para necessidades específicas, também se encaixam nesse grupo. O segundo grupo consiste no reúso de componentes de uma aplicação que variam desde subsistemas até objetos isolados, como por exemplo, um componente que realiza a equivalência de padrões (*pattern-matching system*), desenvolvido como parte de um sistema de processamento de texto, pode ser reutilizado em outro sistema. O terceiro grupo, por sua vez, corresponde ao reúso de classes e funções que implementam uma única funcionalidade.

Vale ressaltar que o reúso de software não se limita a apenas trechos de códigos, mas também abrange análise, padrões, casos de teste, bibliotecas, documentação e frameworks, entre outros artefatos de software. Sendo assim, muitas abordagens têm sido criadas para apoiar o processo de reutilização. Essas técnicas exploram o fato que os sistemas de um mesmo domínio de aplicação possuem similaridades e potencial para reutilização. Na Figura 2 é mostrada a visão geral das possíveis formas de reúso de software, segundo Sommerville (2011).

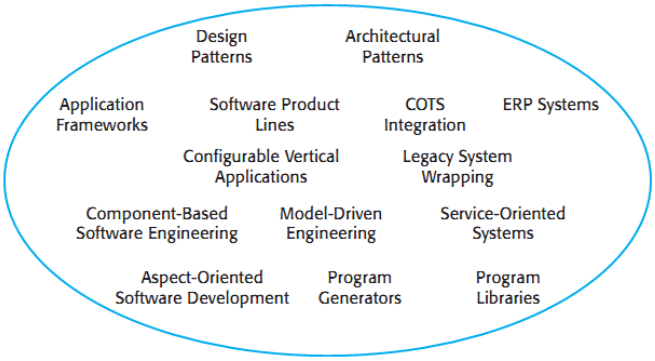


Figura 2: Visão geral das abordagens para o reúso de software (Sommerville, 2011)

O Quadro 1 mostra algumas descrições das abordagens de reúso existentes. A escolha da melhor abordagem deve levar em conta os requisitos necessários para o sistema a ser desenvolvido, a tecnologia e recursos disponíveis, além do conhecimento da equipe de desenvolvimento, podendo-se optar por uma ou mais abordagens para a reutilização de software.

Abordagem	Descrição
Padrões Arquiteturais (*Architectural Patterns*)	Arquiteturas de software que apoiam tipos comuns de aplicações são usadas como base para outras aplicações
Padrões de Projeto (*Design Patterns*)	Abstrações genéricas que ocorrem em todas as aplicações são representadas como padrões de projeto mostrando objetos concretos e abstratos e suas interações
Desenvolvimento baseado em componentes (*Component-Based Software Engineering*)	Sistemas são desenvolvidos por integração de componentes (coleções de objetos) que estejam em conformidade com um modelo de componentes padrão
Frameworks de Aplicação (*Application Frameworks*)	Coleções de classes abstratas e concretas são adaptadas e estendidas para criar aplicações
Empacotamento de sistemas legados (*Legacy Systems Wrapping*)	Sistemas legados são empacotados ("*wrapped*") por meio da definição de um conjunto de interfaces que proveem o acesso a esses sistemas
Sistemas Orientados a Serviços (*Service-Oriented Systems*)	Sistemas são desenvolvidos por meio da utilização de serviços compartilhados, que podem ser providos externamente
Linhas de Produtos de Software (*Software Product Lines*)	Um tipo de aplicação é generalizada em torno de uma arquitetura comum que pode ser adaptada por diferentes clientes
Integração de produtos de prateleira produzido por terceiros (*Integration COTS - Commercial-off-the-shelf*)	Sistemas são desenvolvidos configurando e integrando aplicações existentes
Sistemas ERP (*ERP Systems*)	Sistemas de larga escala que encapsulam funcionalidades de negócios genéricos e regras configuradas para uma organização
Aplicações verticais configuráveis (*Configurable Vertical Applications*)	Sistemas genéricos são projetados de modo que possam ser configurados para necessidades dos clientes específicos do sistema
Bibliotecas de programas (*Programs Libraries*)	Bibliotecas de funções e classes que implementam abstrações usadas com frequência são disponibilizadas para reutilização
Engenharia dirigida por modelo (*Model-driven Engineering*)	O Software é representado como um modelo de domínio e implementação de modelos independentes, o código é gerado a partir desses modelos
Geradores de Programas (*Program Generators*)	Um gerador de sistemas incorpora o conhecimento de um tipo de aplicação, possibilitando gerar sistemas em que o domínio de um modelo de sistema é fornecido pelo usuário
Desenvolvimento de Software Orientado a Aspectos (*Aspect-Oriented Software Development*)	Permite o reuso de funcionalidades transversais que são modularizadas por meio de aspectos, podem ser combinadas em diferentes pontos de uma mesma aplicação quando o programa é compilado, mantendo a separação dessas funcionalidades, evitando o entrelaçamento e espalhamento de código

Quadro 1: Abordagens que visam o reúso de software (adaptado de Sommerville, 2011)

Considerando as abordagens de reúso de software mais utilizadas, mostradas no Quadro 1, nota-se o enfoque de possibilitar que artefatos resultantes de um determinado desenvolvimento possam ser reutilizados para atender posteriormente um conjunto de sistemas. Nesse contexto, tem-se como alternativa o reúso de padrões de software que descrevem soluções para problemas que ocorrem com

frequência no desenvolvimento de software, com o intuito de captar a experiência adquirida pelos desenvolvedores durante anos de prática profissional (Gamma et al., 1993; Braga, 2002). Os padrões fornecem exemplos a serem seguidos e artifícios que podem ser refinados ou estendidos, garantindo a uniformidade na estrutura do software, aumentando a produtividade e facilitando a manutenção (Gall et al., 1996).

A utilização de uma arquitetura de referência também constitui outra forma de reúso de software e pode ser definida como um modelo de referência mapeado para elementos de software e os fluxos de dados entre eles, ou seja, a arquitetura de referência mapeia as funcionalidades em componentes de software (Bass et al., 2003). Uma arquitetura de referência oferece meios concretos para que diferentes produtos possam ser instanciados. Para isso, o engenheiro de domínio deve separar os principais elementos do domínio da arquitetura de referência, pois com uma arquitetura de referência definida pode-se detectar falhas e erros antes da implementação.

Os frameworks constituem uma forma de reúso de software amplamente empregada no desenvolvimento orientado a objetos. Segundo Fayad e Schmidt (1997), um framework pode ser compreendido como uma estrutura que deve ser preenchida com as especificações de uma determinada aplicação para atender uma necessidade particular, ou seja, um framework possui em sua arquitetura duas partes: a parte fixa e a variável. A segunda possibilita a instanciação e, consequentemente, a utilização do framework por aplicações distintas.

Além da utilização de frameworks, tem-se a abordagem conhecida como Linha de Produtos de Software ou Família de Produtos que consiste em um processo sistemático de geração de aplicações a partir de um conjunto de funcionalidades compartilhadas, visando atender um domínio comum (Clements e Northrop, 2002). Por meio desse processo, funcionalidades podem ser reutilizadas por diversas aplicações e as alterações realizadas na linha de produtos são aplicadas a todas as aplicações que forem derivadas.

Nota-se que Linha de Produtos de Software contém uma arquitetura que é compartilhada por todos os produtos derivados e permite aos engenheiros de software construírem programas contendo apenas as funcionalidades necessárias, dentre as fornecidas pela LPS, diferente da utilização de frameworks que apresentam menor flexibilidade de composição e decomposição de features, pois

uma aplicação resultante da instanciação de um framework, por exemplo, poderá conter módulos do framework que não serão utilizados.

A composição de membros em uma LPS resulta em uma aplicação executável, enquanto que para obter uma aplicação completa com apoio de frameworks, deve-se realizar o processo de instanciação. O desenvolvedor deve utilizar os "pontos variáveis" e "métodos ganchos" (*hook methods*) disponíveis, adaptando e/ou estendendo determinadas classes do framework, para criar uma aplicação que atenda aos requisitos necessários, inserindo informações e assim obter a arquitetura completa de uma aplicação. Por exemplo, para utilizar o framework Hibernate, inicialmente é necessário informar o driver do banco de dados, *host* (caminho da rede onde se encontra o banco de dados), a porta utilizada, o nome do banco de dados, o usuário e a senha.

Como observado, cada abordagem possui um processo de reúso definido e específico. Nesse contexto, Ezran et al.(2002) classificam o processo de reúso quanto a estratégia, em dois tipos: sistemático e não sistemático. O primeiro tipo de reúso consiste na utilização de um planejamento que considere a exploração de similaridades dos requisitos e/ou da arquitetura entre as aplicações, visando promover melhorias em termos de produtividade, qualidade e desempenho de negócios. O segundo é dependente da iniciativa e conhecimento individual, sujeito a pouco ou nenhum planejamento, gestão e controle. Logo, o mais provável é que o reúso não sistemático seja caótico em seus efeitos, alimentando a cultura do "heroísmo individual" e o "combate de incêndios" (*fire-fighting*), aumentando os problemas ao invés de reduzí-los.

Sommerville (2011) define o processo de reúso quanto ao desenvolvimento em duas categorias: o desenvolvimento para reúso e o desenvolvimento com reúso. No primeiro, o objetivo é produzir um ou mais elementos reutilizáveis e torná-los disponíveis. Para isso, deve-se adaptar e ampliar os elementos específicos de aplicação para torná-los mais genéricos e, portanto, reutilizáveis. No segundo, o Engenheiro de Aplicação pode não conhecer os elementos disponíveis em sua totalidade, sendo assim, é necessário selecionar apenas os elementos adequados e projetar o sistema para fazer uso mais eficaz dos mesmos.

De modo geral, o reuso é possível em vários níveis, desde funções simples até sistemas completos, e a distinção desses dois tipos de desenvolvimento é um

dos pontos fundamentais das abordagens conhecidas como Frameworks e Linha de Produtos de Software, descritas nas Seções 2.2.1 e 2.2.2, respectivamente.

2.1.1 Frameworks Orientados a Objetos

A adoção do paradigma orientado a objetos, no cenário de desenvolvimento de software em meados da década de 80, impulsionada pela linguagem *Smalltalk*, estimulou a busca por níveis de reutilização, modularidade e extensibilidade que pudessem ser mais efetivos. Sendo assim, iniciou-se o processo de construção de Frameworks Orientados a Objetos (FOO) que acrescentavam às bibliotecas de classes de objetos, os relacionamentos e interações entre as classes, constituindo uma nova forma de reúso com as pretensões esperadas.

A utilização de frameworks proporciona não apenas reutilizar seus componentes isolados, mas uma arquitetura parcialmente completa de um domínio específico. Os frameworks podem ser definidos sob vários pontos de vista. O Quadro 2 possui algumas definições para frameworks quanto a sua estrutura.

Johnson (1991) e Gamma et al. (1995)	Um framework corresponde a um conjunto de objetos que colaboram entre si, visando atender um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação.
Fayad e Schmidt (1997)	Os frameworks representam uma estrutura formada por blocos pré-fabricados de software, que colaboram para uma determinada finalidade dentro de um domínio de um subsistema de aplicações e esta estrutura pode ser adaptada, estendida ou reutilizada.
Govoni (1999)	Um framework consiste em um conjunto de classes, interfaces e padrões dedicados a favorecer a solução de um grupo de necessidades através de uma arquitetura flexível e extensível.
Pinto (2000)	Framework é definido como um software parcialmente completo e projetado para ser reutilizado, constituído por uma coleção de classes abstratas e concretas, intermediadas por interfaces. Para utilização de frameworks há pontos de extensão (*hot-spots*) em que devem ser realizadas adaptações para seu funcionamento.

Quadro 2: Definições de frameworks quanto à estrutura

Os frameworks também podem ser definidos sob o ponto de vista do propósito. O Quadro 3 mostra algumas definições quanto a essa perspectiva.

Johnson e Foote (1988)	Define-se um framework como uma aplicação semi-completa e reutilizável que ao ser especializada produz aplicações customizadas.
Johnson (1997)	Um framework pode ser definido como um esqueleto de uma aplicação, que pode ser instanciado por um desenvolvedor de aplicações. Frameworks fazem com que desenvolvedores não precisem começar do nada, sempre que constroem uma nova aplicação.
Fayad e Johnson (2000)	Um Framework pode ser definido como uma pequena aplicação completa contendo uma estrutura estática e outra dinâmica, desenvolvidas para resolver um conjunto restrito de responsabilidades.

Quadro 3: Definições de frameworks quanto ao propósito

Apesar de haver várias definições para frameworks, nota-se que os conceitos não são contraditórios. De forma geral, os frameworks são ferramentas que promovem o reúso em domínios específicos e para isso, fornecem uma estrutura genérica para construção de aplicações, de forma que essa estrutura deve ser adaptada para uma aplicação específica. Os frameworks são capazes de servir como “esqueletos extensíveis”, em que os métodos fornecidos pela aplicação em desenvolvimento, tornam-se vinculados ao código genérico definido pelo framework (Johnson e Foote, 1988).

Além das definições mencionadas, os frameworks podem ser classificados em frameworks de aplicação (Fayad et al., 1999a; Fayad e Johnson, 2000) ao atender a domínios de aplicação mais amplos e fundamentais para atividades de negócios em empresas. Segundo Fayad et. al. (1999a), apesar desses frameworks demandarem maior esforço ao serem desenvolvidos, permitem maior produtividade quanto ao desenvolvimento de aplicações, pois encapsulam o conhecimento sobre o domínio no qual a empresa atua.

Mesmo não sendo uma classificação excludente, ou seja, à parte das demais classificações, existem os frameworks de componentes, que de acordo com Szyperski (1997), oferecem apoio a componentes que adotam um determinado modelo e permitem que instâncias desses componentes sejam “plugadas” ao framework. Esse tipo de framework coordena a interação entre instâncias de componentes. Uma aplicação pode utilizar em sua arquitetura um único framework de componentes e esse pode agregar gradativamente outros componentes externos e até mesmo outro framework de componentes, enquanto que os frameworks de

aplicação definem uma solução parcialmente completa que possibilita a geração de várias aplicações de um domínio específico.

Segundo Fayad et al. (1999b), considerando o escopo, isto é, a abrangência de sua aplicação em um determinado domínio, os frameworks de aplicação podem ainda ser subdivididos em três subcategorias: (a) frameworks de infraestrutura de sistemas, (b) frameworks de integração de *middleware* e (c) frameworks de aplicações empresariais.

Frameworks de infraestrutura de sistemas simplificam o desenvolvimento de sistemas, como por exemplo: frameworks de comunicação, frameworks de interface gráfica, sistemas operacionais para *mobile* e ferramentas para processamentos de linguagens. Frameworks de integração de *middleware* são geralmente utilizados para realizar integração entre aplicações e componentes distribuídos, como: Java RMI (*Remote Method Invocation*) e framework ORB (*Object Request Broker*), enquanto que os frameworks de aplicações empresariais visam atender um domínio específico de aplicações, como telecomunicações, financeiro, administrativo, entre outros. Esses frameworks são mais custosos de serem desenvolvidos quando comparados aos frameworks de integração de *middleware* e de infraestrutura, pois a construção de frameworks de aplicações corporativas envolve o acompanhamento de especialistas do domínio de aplicações.

Pree e Koshimies (2000) apontam a utilização de outro tipo de framework, os denominados *framelets,* que são frameworks menores, com poucas classes, não assumem o controle de uma aplicação e têm uma interface claramente definida. Assim como os demais FOOs, um *framelet* pode ser especializado por meio de herança e composição. A utilização de vários *framelets* relacionados a um mesmo domínio pode ser considerado uma alternativa à utilização de frameworks muito extensos.

Além dessas classificações, para Yassin e Fayad (2000) os frameworks também podem ser classificados quanto à forma de estendê-los, ou seja, a forma de reúso para criação de novas aplicações, que podem ser três tipos: (1) framework caixa branca ou *white-box*, (2) framework caixa preta ou *black-box* e (3) framework caixa cinza ou *gray-box*.

Frameworks do tipo caixa branca são instanciados por meio de herança, isto é, devem-se criar subclasses das classes abstratas fornecidas pelo framework para criar aplicações, ou por meio de redefinição de “métodos ganchos”, ou *hook-*

methods que são disponibilizados (Pree, 1995). Esses métodos possuem uma implementação possível de ser redefinida na subclasse.

Uma propriedade importante dos frameworks caixa-branca é a "inversão de controle", também conhecida por princípio de *Hollywood – "Don't call us, we'll call you"*, em uma analogia com a forma como são feitos os negócios do cinema. Assim como são os estúdios que chamam o ator para o filme, é o framework que chama a implementação feita pelo desenvolvedor, e não o contrário.

Para Johnson e Foote (1988) com a inversão de controle, o framework assume o papel do programa principal em coordenar e sequenciar a atividade da aplicação, oferecendo aos frameworks o poder de servir como se fossem "esqueletos" de aplicação extensíveis. Segundo Johnson (1997), essa é uma das características principais que tornam os frameworks diferentes de bibliotecas, uma vez que as bibliotecas fornecem um conjunto de funções que podem ser usadas, e os frameworks podem realizar o papel da aplicação principal no sequenciamento das atividades.

Com essa característica, os frameworks incorporam um comportamento extensível através da implementação de suas interfaces, ao criar subclasses ou "plugar" módulos. Depois, é o próprio código do framework que usa essa implementação específica que lhe foi entregue, ou seja, os métodos definidos na aplicação são chamados a partir do framework ao invés de serem por meio do código da aplicação. A inversão de controle consiste na mudança do fluxo de controle de um programa, em que ao invés do programador determinar quando um procedimento será executado, ele apenas determina qual procedimento deve ser executado para um determinado evento.

Em frameworks do tipo caixa preta, o desenvolvedor combina diversas classes concretas existentes no framework para obter a aplicação desejada. Frameworks caixa preta são considerados mais fáceis de utilizar que os frameworks caixa branca, uma vez que o segundo exige que o desenvolvedor, no processo de instanciação do framework, compreenda a estrutura interna do framework. Por outro lado, os frameworks caixa preta não exigem tanto conhecimento sobre a estrutura interna do framework, porém são menos flexíveis, uma vez que o desenvolvedor deverá selecionar uma das variabilidades possíveis que são fornecidas pelo framework. Já os frameworks do tipo caixa cinza, ou *gray-box*, possuem características dos frameworks caixa branca e preta, de forma a evitar desvantagens

de ambos, permitindo extensibilidade sem a necessidade de expor determinadas informações.

Bushmann et al. (1996) ressaltam que a instanciação de frameworks envolve a composição e criação de subclasses de classes existentes. Um framework de aplicação consiste em “pontos fixos”, ou *frozen spots* e “pontos de especialização”, ou *hot spots*.

Os *Frozen spots* definem uma arquitetura geral de um sistema de software que considera os componentes básicos e seus relacionamentos, ou seja, são partes do framework que permanecem fixas em qualquer instanciação, enquanto que os *hot spots* representam as partes modificáveis e específicas do framework para possibilitar o acoplamento com o código base da aplicação. Os *hot spots* são projetados para serem genéricos. Sendo assim, podem ser adaptados de acordo com a necessidade de uma aplicação em desenvolvimento. Schmid (1997) considera aspectos variáveis em um domínio de aplicação sendo *hot spots*, e que diferentes aplicações a partir de um domínio diferem no que diz respeito a algum (ou pelo menos um) dos *hot spots.*

A parte (a) da Figura 3 representa um framework tipo caixa branca com um *hot spot* R. Para a utilização desse framework é necessário fornecer a implementação representada por R_3 a esse *hot spot*. Vale ressaltar que na parte(a) da Figura 3, as linhas tracejadas delimitam o framework do código necessário para sua instanciação, ou seja, o código de instanciação não é disponibilizado pelo framework, por isso o desenvolvedor deve utilizar os “métodos gancho” fornecidos pelo *hot spot* R e estender classes abstratas para introduzir informações específicas da aplicação e assim utilizar o framework.

A parte (b) da Figura 3 ilustra um framework caixa preta que possui da mesma forma como o framework caixa branca, um *hot spot* R. Este framework possui as alternativas, R_1, R_2 e R_n para a implementação do *hot spot* R. Observa-se que as variabilidades são fornecidas pelo framework, mas são as únicas possíveis. O engenheiro de aplicação pode escolher uma dessas alternativas possíveis, já na parte (a) da Figura 3, R_3 não pertence ao framework, mas qualquer alternativa de implementação do *hot spot* seria possível.

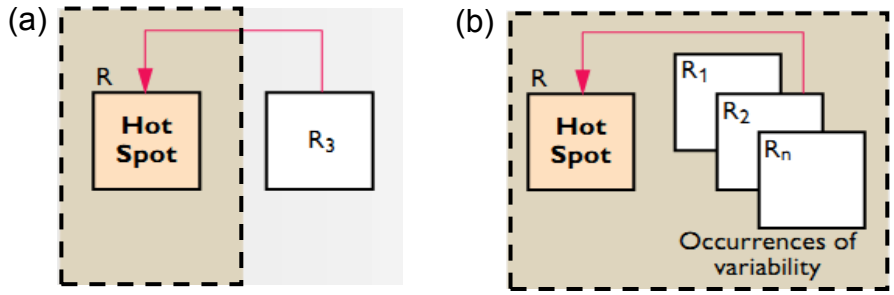


Figura 3: *Hot spot* em frameworks caixa branca (a) e caixa preta (b) (adaptado de Schmid, 1997)

Nota-se que os frameworks caixa branca são mais fáceis de projetar, pois não exigem do desenvolvedor prever todas as alternativas de implementação possíveis. Frameworks caixa preta obrigam essa previsão, logo são mais difíceis de projetar. Todavia, são mais fáceis de utilizar, bastando selecionar a implementação desejada e combinar as classes concretas fornecidas pelo framework, enquanto que em frameworks caixa branca é necessário realizar as implementações adequadas, pois as alternativas possíveis não são fornecidas.

De forma geral, os frameworks fornecem um conjunto de funcionalidades que podem ser reutilizadas por diversas aplicações, evitando a necessidade de implementar uma série de recursos previamente disponibilizados, favorecendo a produtividade e a entrega de software em prazos menores.

Apesar desses benefícios, a manutenção de um framework não é tão simples, uma vez que as aplicações derivadas do framework necessitam passar por um processo de manutenção correspondente, para tornarem-se compatíveis com a nova versão framework, ou então o framework deverá ser dividido em várias versões, o que pode implicar em maior esforço para mantê-lo (Bosch et al., 1999). Logo, nem sempre é viável corrigir todas as aplicações derivadas de um framework, em razão do alto custo e, muitas vezes, ao grande (ou até mesmo ignorado) número de aplicações existentes, considerando o alto risco de introduzir novos erros nessas aplicações (Braga, 2002).

Além dessas dificuldades, os frameworks mostram-se inflexíveis quanto à composição e decomposição de suas features, uma vez que as aplicações instanciadas de frameworks, mesmo utilizando apenas parte das funcionalidades, levam consigo todas as classes disponibilizadas pelos frameworks (Codenie et al., 1997), dificultando a posterior manutenção das aplicações, por não possuírem somente os recursos necessários.

2.1.2 Linha de Produtos de Software

A produtividade obtida por meio da reutilização de software tem sido evidenciada com a adoção de abordagens que favorecem níveis mais efetivos de qualidade e menor tempo necessário para o desenvolvimento de aplicações.

Uma dessas abordagens é denominada Linha de Produtos de Software (LPS) ou Família de Produtos (FP) que trata do desenvolvimento de aplicações por meio de um conjunto comum de características (*features*) pré-desenvolvidas ao invés de desenvolver aplicações separadamente. Segundo Clements e Northrop (2002), uma LPS é constituída por um conjunto de sistemas de software que compartilham um conjunto de características comuns e gerenciadas, que satisfazem às necessidades específicas de um determinado segmento de mercado ou um objetivo, e que são desenvolvidos a partir de um conjunto comum de ativos (*assets*), de maneira prescrita, ou seja, cada produto é desenvolvido de forma sistemática a partir do conjunto de artefatos compartilhados pela LPS.

Batory et al. (2002) apud Czarnecki e Eisnecker (2000) tratam os termos LPS e FP como conceitos distintos porém, com certas similaridades. Uma LPS compartilha funcionalidades para geração de aplicações visando atender um domínio comum, enquanto que uma FP atende a domínios diferentes e suas aplicações derivadas podem ser integradas gerando uma suíte de programas. As aplicações geradas a partir de uma FP possuem capacidades específicas de seu respectivo domínio, como por exemplo, o Microsoft Office[4] que inclui os programas com capacidades diferentes: Excel (planilhas), Word (editor de textos), Access (banco de dados) e etc.

De modo geral, uma LPS possui uma arquitetura flexível e reusável com o objetivo de permitir a customização de aplicações que pertencem a um domínio específico, proporcionando uma série de benefícios (Clements e Northrop, 2002), como: (a) produtividade em larga escala, (b) redução do custo e tempo de desenvolvimento, pois os produtos derivados da LPS podem ser entregues rapidamente, (c) se manter presente no mercado de software, uma vez que as modificações e o gerenciamento da LPS são realizados de forma organizada e centralizada, (d) satisfação dos clientes, pois considerando que uma LPS sofre

[4] http://office.microsoft.com/

diversas evoluções, possivelmente suas funcionalidades foram validadas e testadas, (e) melhores níveis de reutilização, (f) customização em massa das aplicações, pois as alterações realizadas na LPS não aplicadas a todas as aplicações derivadas, (g) muitas vezes, compensa a não contratação de novos engenheiros de software, uma vez que uma LPS consistente e organizada permite que um número reduzido de engenheiros de software a manipulem e apliquem as devidas manutenções e (h) níveis efetivos de qualidade, obtida por meio de técnicas de implementação de variabilidades.

2.1.2.1 Modelo de Features

No desenvolvimento de LPS é necessário identificar as similaridades e as diferenças entre os produtos em termos de requisitos. Essas funcionalidades comuns e variáveis são chamadas de features. Uma feature (Czarnecki e Helsen, 2006) é uma propriedade ou uma funcionalidade de uma família de sistemas que é relevante para algum dos interessados no seu desenvolvimento. Dentre diversas formas para o desenvolvimento e representação de LPS, o método FODA (*Feature-Oriented Domain Analysis*) proposto por (Kang et al., 1990) é um dos mais conhecidos. Com base nesse método, as funcionalidades cobertas pelo domínio de uma LPS, podem ser modeladas com um nível de abstração satisfatório, favorecendo a modularidade na construção e customização de programas derivados (Batory et al., 2002). Esse tipo de modelagem tem sido amplamente adotada pela comunidade, na qual diversas extensões têm sido propostas (Benavides et al, 2005; Barreiros e Moreira, 2011).

Com o método FODA, podem-se construir modelos de features para representar de forma compacta todos os produtos de uma LPS em termos de features. Nesses modelos, as features devem estar organizadas com propriedades e relacionamentos apropriados. A partir desse modelo, pode-se compor features gerando diferentes combinações. Cada combinação indica um possível produto que pode ser derivado, ou seja, uma aplicação concreta. Essa combinação pode ser denominada “configuração”. Uma configuração válida garante que todas as restrições e dependências criadas no modelo foram consideradas. Essa alternativa apoia não apenas o processo de desenvolvimento de LPS, como também a construção de artefatos referentes à documentação, arquitetura e trechos de código.

Na Figura 4 ilustra-se um modelo de features, segundo a notação do método FODA, de uma LPS que permite gerar aplicações do domínio de comércio eletrônico (E-Shop). Nessa figura, visualizam-se a representação dos relacionamentos entre uma feature "pai" e suas features "filhas", ou subfeatures. Note que as features "Catálogo", "Pagamento" e "Segurança" são obrigatórias, assim devem ser incluídas por todas as configurações dos produtos, enquanto que a feature "Pesquisa", por ser opcional, representa uma opção de escolha, logo pode ou não pertencer às aplicações derivadas. As features "TransferênciaBancária" e "CartãodeCrédito" estão associadas a feature "Pagamento". Esse relacionamento é realizado por meio de um conector "ou", logo, pelo menos uma dessas subfeatures pode ser selecionada. Além disso, as features "Alta" e "Padrão" estão também associadas, mas por um tipo diferente de relacionamento, em que somente uma dessas features pode ser selecionada. Por exemplo, a seleção da feature "Alta" impede que a feature "Padrão" faça parte da mesma configuração e vice-versa. Esse relacionamento é do tipo alternativo ("xor"), isto é, mutuamente exclusivo.

Além dos relacionamentos de parentesco entre as features, restrições entre as features também podem ser utilizadas, como por exemplo: **X *requires/implies* Y** – a seleção de X implica na seleção de Y; **X *excludes* Y** – X e Y não podem fazer parte do mesmo produto. Como exemplo, na parte inferior da Figura 4 há uma regra em que a seleção da feature "CartãodeCrédito" implica na seleção da feature "Alta", indicando que para esse tipo de pagamento, o nível de segurança deve ser alto.

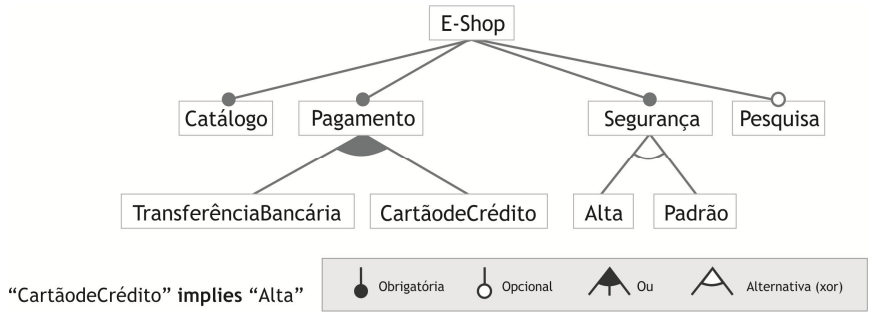


Figura 4: Exemplo de Modelo de Features

Sob a perspectiva semântica (Schobbens et al., 2006), o modelo de features possui um lado formal em que a seleção de features deve respeitar. Por exemplo, se uma feature f_1 é uma subfeature obrigatória de f_2, formalmente, a expressão para representar isso, pode ser definida como $f_1 \Leftrightarrow f_2$. O conector de equivalência material

(⇔) indica que ao selecionar f_1, a feature f_2 também é selecionada. No Quadro 4 são descritos os principais tipos de relacionamentos entre features e suas respectivas representações formais, demonstrando a semântica do modelo de features.

Relacionamento entre features	Representações formais
∂ *é uma feature raiz no modelo*	∂
$\int 1$ *é uma subfeature opcional de* $\int$	$\int 1 \Rightarrow \int$
$\int 1$ é uma subfeature obrigatória de $\int$	$\int 1 \Leftrightarrow \int$
$\int 1,...,\int n$ são subfeatures alternativas de $\int$	$\left(\int 1 \vee ... \vee \int n \Leftrightarrow \int\right) \wedge \bigwedge_{i<j} \neg\left(\int i \wedge \int j\right)$
$\int 1,...,\int n$ são subfeatures "or" de $\int$	$\int 1 \vee ... \vee \int n \Leftrightarrow \int$
$\int 1$ *excludes* $\int 2$	$\neg(\int 1 \wedge \int 2)$
$\int 1$ *requires* $\int 2$	$\int 1 \Rightarrow \int 2$

Quadro 4: Semântica de um Modelo de Features (baseado em Schobbens et al., 2006)

A utilização de features favorece a escalabilidade das LPS (Batory et al., 2002), pois facilita a adição de novas features, ampliando as variabilidades que podem ser fornecidas. Desse modo, existem várias estratégias de implementação de features, como: diretivas *ifdef*, *Collaboration-based design*, *Mixin Layers*, Programação Orientada a Aspectos (POA), templates, entre outras.

Loesch e Ploedereder (2007) e Kästner et al.(2008) apontam a importância da obtenção de um nível maior de granularidade na implementação de *features*. Para isso, utilizam diretivas de pré-processamento *ifdef* e *endif*, ou seja, anotações explícitas no código fonte para definir as possíveis *features* de um código legado. Com o código fonte contendo as anotações apropriadas, os fragmentos de código são associados as suas features correspondentes. A utilização dessas diretivas é voltada ao tratamento de *features* de granularidade fina. As abordagens de anotação como essas diretivas, podem implementar essas extensões, mas por outro lado, podem apresentar problemas de legibilidade por "ofuscar" o código fonte.

Além da estratégia *ifdef*, tem-se a abordagem *Collaboration based-design,* que parte do princípio de que na orientação a objetos, os objetos são entidades encapsuladas que raramente são autossuficientes. Dessa forma, a semântica de um

objeto pode ser definida em grande parte por sua relação com outros objetos. A interdependência dos objetos pode ser expressa por meio de colaborações. Uma colaboração é um conjunto de objetos e um protocolo (isto é, um conjunto de comportamentos permitidos) que determina a forma como esses objetos interagem. A parte de um objeto relacionada com o protocolo de uma colaboração é chamada papel do objeto. A abordagem *Collaboration-based design* é baseada na composição dessas colaborações. Dessa maneira, cada objeto de uma aplicação representa uma coleção de papéis que descrevem as ações sobre dados comuns. Cada colaboração corresponde a uma *feature* que encapsula relacionamentos de seus objetos correspondentes (Batory et al., 2000 apud Reenskaug, et al., 1992; VanHilst e Notkin 1996; Smaragdakis e Batory 1998).

A utilização de *Mixin layers* corresponde à outra estratégia de implementação de *features* (Batory et al., 2000 apud Bracha e Cook, 1990). Uma *Mixin* corresponde a uma classe, cuja superclasse é especificada por um parâmetro. *Mixins* fornecem a capacidade de criar hierarquias de herança quando compostas. As *mixin-layers* são uma forma de implementar colaborações, ou seja, uma *mixin* com classes aninhadas (*nested classes*), onde cada classe aninhada corresponde a um papel de uma colaboração. As *mixin-layers* podem ser implementadas por meio de templates *Jak*. A linguagem *Jak* consiste em um superconjunto extensível Java que suporta meta-programação, ou seja, *features* que permitem aos programas Java escreverem outros programas em linguagem Java (Batory et al. 1998). A Figura 5 ilustra um exemplo de definição de uma *mixin-layer*. A *mixin* M tem n+1parâmetros: um parâmetro S que define a superclasse de M, acrescido de n parâmetros adicionais $\{r_1 ... r_n\}$ que definem as classes de papéis de colaboração que são refinadas. Uma *mixin* pode ser considerada uma feature contendo classes que são refinadas (subclasses) para atender um determinado requisito da aplicação.

```
class M <AnyClass S, AnyClass r1, ... AnyClass rn>
extends S {
   class role1 extends r1 { ... }
   ...
   class rolen extends rn { ... }
   ...
   // additional non-refining classes (if any)
}
```

Figura 5: Exemplo de implementação de mixin-layer (Batory et al., 2000)

A Programação Orientada a Aspectos (POA) (Kiczales et al., 1997) (ver Seção 3.1), também pode ser empregada na implementação de features. Apesar de essa técnica visar à modularização de interesses transversais, a mesma também pode ser utilizada para modularizar interesses funcionais que não necessariamente estejam espalhados e entrelaçados em um determinado sistema. Assim, chamadas e refinamentos de métodos podem ser modularizados em unidades aspectuais que posteriormente poderão ser agrupadas em features. Por exemplo, uma feature opcional desenvolvida utilizando essa técnica, ao fazer parte de um dado produto da configuração de um modelo de features, poderá introduzir estaticamente ou dinamicamente vários comportamentos nesse produto, pois os mecanismos de composição abstratos previstos na POA permitem esse tipo de composição. Entretanto, a implementação de features de granularidade fina com POA pode se tornar complexa e implicar em uma quantidade exponencial de unidades aspectuais para tratar todas as combinações entre as features.

Além dessas técnicas, ferramentas para geração de código fonte a partir de modelos também podem ser empregadas. No Desenvolvimento Dirigido a Modelos (MDD) (ver Seção 3.2), a partir de um modelo abstrato pode-se obter código fonte, por meio de um processo automatizado (Schmidt, 2006). Dentre as maneiras de utilizar MDD para que uma determinada configuração do modelo de features seja transformada em código, pode-se criar um modelo que reflete a disposição das features e também desenvolver templates.

Nesses templates, todo o código que poderá ser gerado no final do processo de transformação deve ser analisado no sentido de identificar quais fragmentos colaboram com a implementação de cada feature. Com essa identificação, os trechos de código podem ser associados às features presentes no modelo. Essa associação é realizada por meio de "*tags"* específicas da linguagem de especificação do gerador utilizado. Uma das vantagens de se utilizar essa técnica em comparação com POA, é que a implementação tanto de features de granularidade grossa quanto as de granularidade fina pode ser realizada da mesma forma, isto é, inserindo *tags*. No caso da POA, para tratar features de granularidade fina, torna-se necessário, dentre outras atividades, capturar argumentos dos métodos e entrecortar construtores. Isso pode se tornar complexo em sistemas de grande porte. Para modularizar features de granularidade grossa com POA, pode-se utilizar *inter-type declarations* que é relativamente mais simples. No caso da

utilização de templates, a combinação entre as features torna-se “mais visível para o desenvolvedor”, pois o código presente nos templates conterá todas as chamadas e refinamentos previstos e o código gerado com base nas features selecionadas dá-se em tempo de projeto, diferente de utilizar POA, uma vez que certos comportamentos poderão ser inseridos em tempo de compilação.

A POA e a utilização de templates no contexto de MDD foram discutidos sucintamente nesta seção, sob o ponto de vista da implementação de features. Vale ressaltar que elas são empregadas amplamente neste trabalho. Dessa forma, no capítulo 3, tais técnicas são discutidas com mais profundidade.

2.1.3 FeatureIDE

O FeatureIDE é um *plug-in* baseado na IDE Eclipse e apoia o Desenvolvimento de Software Orientado a Feature - *Feature-Oriented Software Development* (FOSD). Esse *plug-in* é *open-source*, disponibilizado sob a licença GPL[5] e oferece suporte às fases do FOSD no contexto de LPS, tais como: análise de domínio, análise de requisitos, implementação e geração de software. Distintas técnicas de desenvolvimento de LPS estão integradas, como: Programação Orientada a Features, POA entre outras. Além disso, o *plug-in* fornece apoio para implementação com: AHEAD, FeatureC++, FeatureHouse, AspectJ, DeltaJ, Munge e Antenna (Thüm et al., 2013).

De modo geral, esse *plug-in* possui um editor de modelo de features que pode ser utilizado em modo gráfico ou em modo texto, uma vez que um modelo do FeatureIDE consiste em um arquivo *.xml*. Além de um editor de modelo, pode-se também implementar restrições por meio de um assistente. Esse assistente, com base em uma verificação sintática e semântica conduz a criação tanto do modelo de features quanto das restrições de maneira coerente. Há também um editor de configuração que permite selecionar features do modelo e derivar produtos.

A Figura 6 ilustra um modelo de features desenvolvido no FeatureIDE. Trata-se do mesmo apresentado na Figura 4, incluindo a regra entre as features “CartãodeCrédito” e “Alta”, além da criação de uma configuração. A parte (1) da figura ilustra o modelo de features, com os devidos relacionamentos e propriedades

[5] http://www.gnu.org/licenses/gpl.html

das features. Na parte (2), o assistente para criação de restrições do modelo. Note que as restrições podem ser construídas com o apoio de vários operadores, tais como: *not*, *and*, *or*, *implies* entre outros. Por fim, a parte (3) ilustra a escolha de features do modelo, ou seja, uma configuração. Note que a seleção da feature "CartaodeCredito" implica automaticamente na seleção da feature "Alta", logo fazem parte da mesma configuração. Isso indica que a restrição visualizada na parte (2) está sendo considerada. Podemos visualizar também na parte (3), que à medida que as features são selecionadas, é apresentado o número de possibilidades de combinação de features na parte superior.

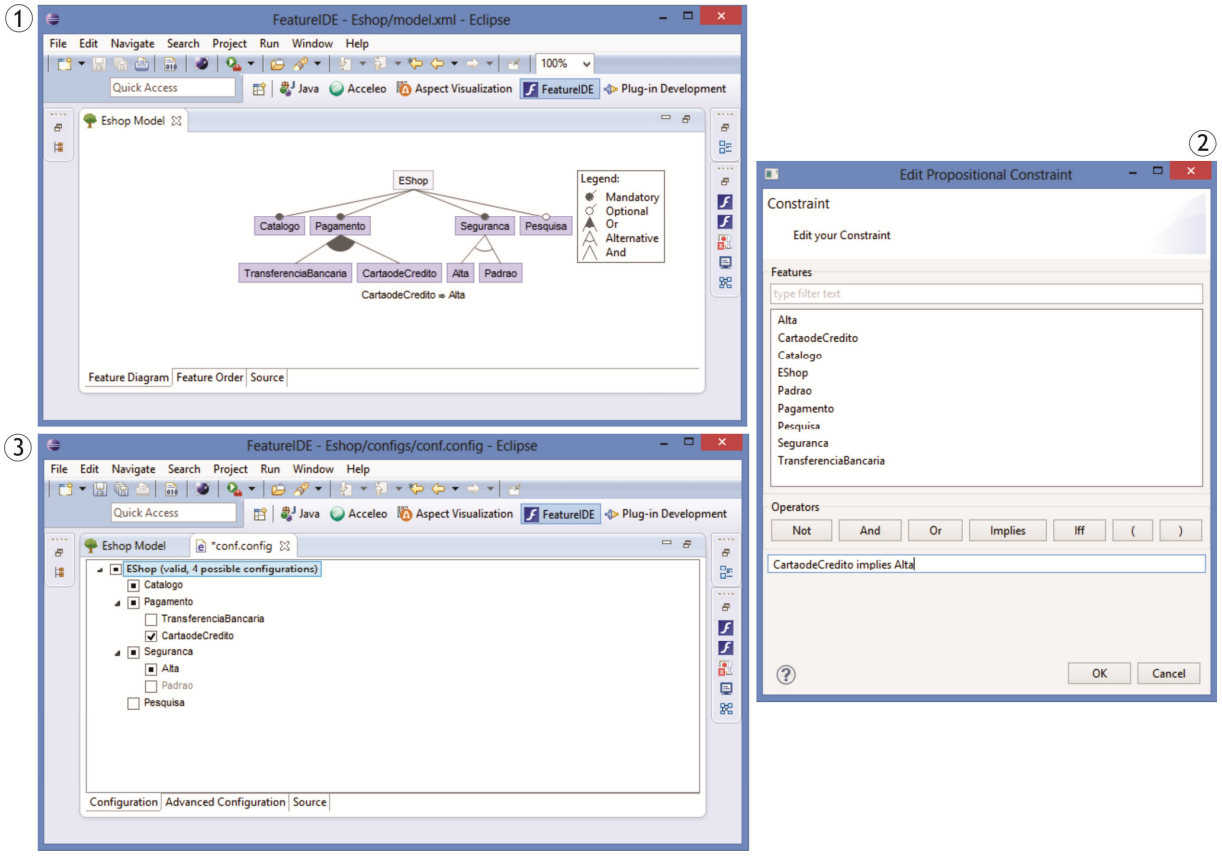


Figura 6: Utilização do FeatureIDE

2.2 Considerações Finais

O reúso de software é uma prática importante no cenário de desenvolvimento de software por proporcionar maior produtividade, qualidade e a redução do tempo para entregar um software funcional cumprindo as expectativas de seus usuários.

Diante disso, muitas abordagens foram criadas com finalidades bem específicas, como as LPS e os Frameworks. As conhecidas LPS atendem um domínio específico, permitem que um desenvolvedor selecione um conjunto de features e por meio de um mecanismo de composição, aplicações prontas podem ser derivadas e posteriormente, fornecidas ao usuário final. Os frameworks podem tratar de um interesse específico como, por exemplo, a persistência de dados em bancos relacionais em diversas aplicações. Desse modo, toda preocupação do desenvolvedor em ter que programar completamente essa parte, torna-se responsabilidade do framework. Dentre vários tipos de frameworks, existem aqueles que apoiam um domínio de aplicações mais restrito, os chamados frameworks de aplicação, que ao serem acoplados a um código base e instanciados adequadamente podem levar a aplicações executáveis.

Sob uma visão prática, há diversas técnicas que podem ser aplicadas seja no contexto de LPS ou de frameworks. Assim, no próximo capítulo discutem-se os conceitos ligados ao Desenvolvimento Orientado a Aspectos e o Desenvolvimento Dirigido a Modelos.

Capítulo 3

DESENVOLVIMENTO ORIENTADO A ASPECTOS E DESENVOLVIMENTO DIRIGIDO A MODELOS

Neste capítulo são apresentados os conceitos relacionados ao Desenvolvimento de Software Orientado a Aspectos (DSOA) e Desenvolvimento Dirigido a Modelos (MDD - Model Driven Development). Esse capítulo fornece embasamento para o capítulo 7 referente ao experimento, no qual se empregaram ambas as técnicas. Na seção de DSOA apresentam-se as principais abstrações dessa modalidade de desenvolvimento consideradas no trabalho. Posteriormente na seção de MDD, descreve-se a transformação de modelos em código fonte e por fim, na seção de Considerações Finais apresenta-se uma síntese dos pontos discutidos neste capítulo.

3.1 Desenvolvimento de Software Orientado a Aspectos

O paradigma orientado a objetos permite a construção de sistemas considerando abstrações do mundo real de forma modularizada, por meio de classes e seus relacionamentos, promovendo a reusabilidade e constituindo o paradigma dominante no cenário atual de desenvolvimento de software (Laddad, 2003).

Apesar dessa ampla utilização, o desenvolvimento orientado a objetos possui certas dificuldades quanto à implementação de interesses que naturalmente afetam múltiplas classes, os denominados interesses transversais (*crosscuting concerns*), como: segurança, *logging*, persistência e auditoria, entre outros. Essas limitações

refletem em módulos espalhados e entrelaçados na arquitetura final das aplicações que dificultam a manutenção e evolução.

Segundo Laddad (2003) o entrelaçamento de código, também denominado emaranhamento, é ocasionado quando um módulo em um sistema de software é implementado para lidar simultaneamente com múltiplos interesses do sistema. A Figura 7 representa vários módulos de implementação de um sistema, contendo trechos de código referente a diversos interesses, como: lógica de negócios, persistência e *logging*.

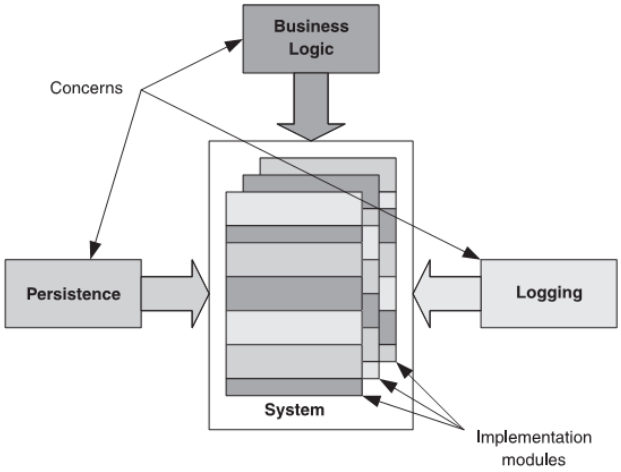


Figura 7: Entrelaçamento de código (Laddad, 2003)

Para Laddad (2003); Winck e Junior (2006), o espalhamento ocorre quando uma única funcionalidade é implementada em vários módulos, ou seja, o código necessário para cumprir um interesse específico propaga-se em classes que tratam de outros interesses. A Figura 8 representa um sistema bancário em que foi implementado o interesse de *logging*, utilizando técnicas orientadas a objetos.

Observa-se que o código referente ao interesse de *logging* está espalhado em outros módulos, ou seja, existem várias invocações aos métodos pertencentes ao módulo *logging*, partindo de módulos que implementam interesses distintos, tais como: banco de dados, caixa eletrônico (ATM - *Automated Teller Machine*) e contabilidade.

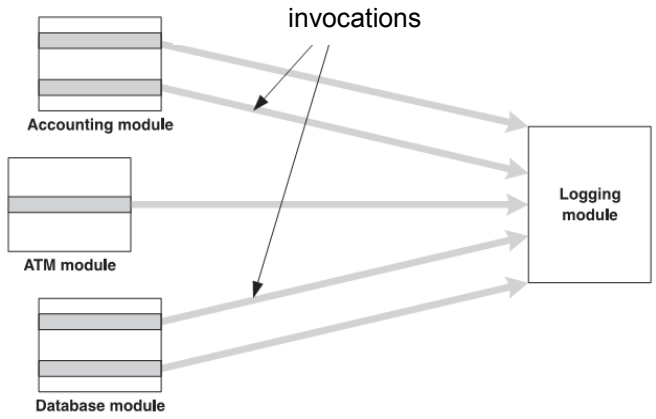


Figura 8: Espalhamento de código (adaptado de Laddad 2003)

A ocorrência de códigos entrelaçados e espalhados reflete diretamente no desenvolvimento de software, causando dificuldades como: (a) alto acoplamento, pois os métodos das classes necessitam admitir métodos das classes que implementam as funcionalidades espalhadas; (b) baixa coesão, uma vez que métodos das classes afetadas possuem instruções que não estão diretamente relacionadas à funcionalidades que implementam; (c) redundância, pois muitos fragmentos de código semelhantes ocorrem em diversos pontos do código fonte e (d) compreensibilidade prejudicada, uma vez que a implementação de interesses transversais deve ser dependente do código espalhado para facilitar o reúso e a manutenção. Por exemplo, o código responsável para implementação de um interesse como o tratamento de exceções, utilizando orientação a objetos, é inseparável do código onde a exceção poderá ser gerada, impedindo que esse interesse seja agrupado em apenas uma unidade.

Considerando essas dificuldades, o princípio da separação de interesses introduzido por Dijkstra (1976), que consiste em decompor o domínio do sistema em partes isoladas para facilitar o entendimento, pode ser aplicado a esse cenário de desenvolvimento de software, visto que cada interesse representa uma parte do domínio do sistema que pode ser considerada como um requisito funcional ou não funcional.

De acordo com Laddad (2003), os interesses de um sistema de software podem ser classificados em dois grupos: interesses de negócio e interesses em nível de sistema, ou interesses transversais. O primeiro compreende interesses funcionais que são relacionados com lógica de negócios, enquanto que o segundo refere-se

aos requisitos que atravessam vários módulos, como chamadas a métodos relacionados com requisitos não funcionais.

Contudo, existem abordagens que visam a separação desses interesses, como a Programação Orientada a Aspectos (POA) proposta por (Kiczales et al.,1997) que contribui para melhorar a modularização dos interesses transversais. Segundo (Kiczales et al., 1997), uma implementação de POA consiste em três elementos principais: linguagem de componentes, linguagem de aspectos e o combinador de aspectos. A linguagem de componentes é responsável por implementar interesses de negócio do sistema de software, por exemplo a linguagem Java. A linguagem de aspectos deve favorecer uma implementação clara de interesses transversais por meio de aspectos. O combinador de aspectos (*aspect weaver*) realiza o "*weaving"*, ou seja, a combinação entre os programas escritos na linguagem de componentes com o código em linguagem de aspectos, por meio de um pré-processamento em tempo de compilação.

As linguagens de aspectos podem ser classificadas em propósito específico ou propósito geral. As linguagens de propósito específico tratam somente de determinados aspectos, possuindo restrições quanto ao uso das linguagens de componentes, enquanto que as linguagens de propósito geral permitem a implementação de qualquer tipo de interesse e podem compartilhar o mesmo ambiente utilizado pela linguagem de componentes, como é o caso da linguagem AspectJ[6], criada pela *Xerox Palo Alto Research Center* em 1997 (Kiczales et al., 2001).

AspectJ complementa a linguagem Java na separação de interesses transversais, acrescentando outros conceitos, como: aspectos (*aspects*), pontos de junção (*join point*s), conjuntos de junção (*point cuts*), adendos (*advices*) e declaração intertipo (*inter-type declaration*). As definições para esses elementos são apresentadas na Seção 3.1.1.

[6] AspectJ. URL: http://www.eclipse.org/aspectj

3.1.1 Abstrações da Programação Orientada a Aspectos

Os aspectos são elementos básicos que permitem isolar interesses transversais, modificando a estrutura estática ou dinâmica de programas orientados a objetos preservando seu comportamento (Laddad, 2003).

A estrutura estática desses programas pode ser alterada por meio de declarações intertipo que permitem a adição de atributos, métodos ou construtores a uma classe, enquanto que, a estrutura dinâmica pode ser modificada em tempo de execução por meio dos conjuntos de junção, os quais são definidos pontos de junção correspondentes aos pontos de execução de um programa que podem ser adicionados comportamentos (adendos). Dessa forma, os trechos de código recorrentes de interesses transversais que ficariam entrelaçados ou espalhados são encapsulados em aspectos.

Os pontos de junção são pontos na execução de um programa de componentes que devem ser definidos para que os aspectos sejam aplicados, ou seja, esses pontos possibilitam o relacionamento entre classes e aspectos, como: chamada e execução de métodos, leitura e alteração de atributos, etc. Os pontos de junção auxiliam o combinador de aspectos na composição dos aspectos com o código orientado a objetos.

Um conjunto de junção é responsável por selecionar pontos de junção, tornando possível a detecção dos pontos do programa em que os aspectos devem atuar, adicionando novos comportamentos e/ou alterando os existentes.

Os adendos são comportamentos que devem ser executados em um ponto de junção referenciado pelo conjunto de junção. Os adendos podem ser classificados em três tipos: anterior, “de contorno” ou posterior (*before*, *around* ou *after*). O primeiro refere-se ao comportamento que deve ser executado antes do ponto de junção. O segundo executa “em volta” do ponto de junção e pode ser utilizado para substituição de trechos de código de um método referenciado pelo ponto de junção e o tipo posterior, por sua vez, executa depois do ponto de junção.

As declarações intertipo possuem a capacidade de modificar a estrutura estática dos módulos de uma aplicação, ou seja, em tempo de compilação as declarações intertipos podem especificar novos atributos, construtores ou métodos nas classes, além de possibilitar a alteração de interfaces e aspectos.

3.2 Desenvolvimento Dirigido a Modelos

No Desenvolvimento Dirigido a Modelos, conhecido como MDD (*Model Driven Development*), diagramas não servem somente para auxiliar os desenvolvedores na fase de implementação, mas também são utilizados como artefatos de entrada para geração de código (France e Rumpe, 2007; Schmidt, 2006).

O MDD tem enfoque na modelagem e na transformação de modelos em artefatos de implementação com o intuito de possibilitar maior produtividade aos processos de desenvolvimento de software. No MDD, os modelos que especificam a aplicação são utilizados para geração de código, total ou parcialmente, para diferentes tecnologias, de forma que as tarefas repetitivas de decodificação para as diversas plataformas fiquem encapsuladas nas transformações, economizando tempo e esforço dos desenvolvedores. Assim, modelos podem ser utilizados como entrada para geradores de código que podem gerar aplicações concretas.

Com o MDD, pode-se reduzir a distância semântica entre o problema e a implementação, gerando código com mais qualidade considerando práticas/padrões eficientes de desenvolvimento, por meio de modelos que possam assumir a complexidade da etapa de implementação (France e Rumpe, 2007). Assim, modelos são utilizados para expressar conceitos de um determinado domínio de forma mais efetiva, enquanto que as transformações desses modelos geram artefatos que refletem uma solução expressa (Schmidt, 2006). Os modelos são formas intuitivas para representação do conhecimento e menos dependentes do código-fonte (Lucrédio, 2009), de forma que podem ser reutilizados facilmente em diferentes projetos.

Nesse contexto, o consórcio *Object Management Group* (OMG) desenvolveu um padrão arquitetural para o MDD que também pode ser considerado como uma especificação, denominado *Model Driven Architecture* (MDA) (OMG, 2003). Essa arquitetura define que o processo de desenvolvimento de software deve ser orientado pela atividade de modelagem do sistema, em nível conceitual, independentemente de uma plataforma de implementação específica e que por meio de transformações nos modelos produzidos durante a modelagem, outros modelos podem ser produzidos com níveis menores de abstração, ligados à implementação.

Dessa maneira, um sistema concreto pode ser gerado automaticamente com base em um modelo conceitual.

MDA é uma abordagem que potencializa a capacidade dos modelos no desenvolvimento de sistemas, promovendo meios para direcionar o curso da compreensão, projeto, construção, implantação, operação, manutenção e modificação (OMG, 2003).

Contudo, a MDA é uma abordagem que utiliza quatro tipos de modelos principais: *Computation Independent Model* (CIM), *Platform-Independent Model* (PIM), *Platform-Specific Model* (PSM) e *Platform Model*. O CIM descreve como um sistema deve se comportar em termos de uma linguagem apropriada para os usuários. O PIM descreve o *Computing Independent Model* (CIM) em termos computacionais, sendo definido com um alto grau de abstração, livre de plataforma. O PIM descreve o sistema de software de uma perspectiva que melhor represente o negócio sendo modelado. O PSM considera detalhes de uma determinada tecnologia a ser utilizada na implementação, enquanto que o *Platform Model* fornece um conjunto de conceitos técnicos que representam diferentes partes que compõem uma plataforma e seus serviços prestados. Além disso, fornece os conceitos que representam elementos para serem utilizados na especialização dessa plataforma por uma aplicação (OMG, 2003).

Para tornar mais clara a relação entre PIM e PSM, a Figura 9 ilustra uma transformação que envolve tais modelos, isto é, a conversão de um PIM para um PSM. Nota-se que o PIM e outras informações são combinados por uma determinada transformação para produzir um PSM. Além disso, a figura é genérica, pois há diversas maneiras em que tal transformação pode ser realizada. No entanto, a ideia é enfatizar que partir de um PIM, um modelo específico para uma plataforma particular pode ser gerado.

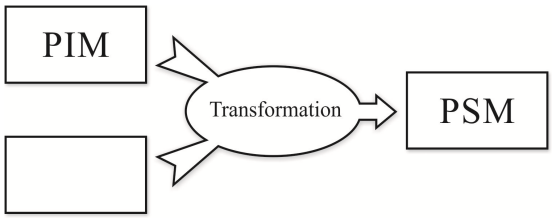


Figura 9: Transformação de Modelos (adaptado de OMG, 2013)

Uma transformação é uma geração automática de um modelo em outro, conforme uma especificação que abrange um conjunto de regras. Vale ressaltar que

a partir de PSMs pode ser gerado código fonte e essa geração não gera somente estruturas básicas, mas pode consistir em uma solução de software definitiva (Kleppe et al., 2003).

Para exemplificar um *Platform Model*, pode-se admitir o Modelo de Componentes CORBA que fornece os conceitos: *EntityComponent, SessionComponent, ProcessComponent, Facet, Receptacle, EventSource* e outros. Esses conceitos são utilizados no uso específico na plataforma de componentes CORBA (*CORBA Component platform - CCM*) por uma aplicação (OMG, 2003).

O MDD pode ser adaptado para diversas finalidades. Por exemplo, a Figura 10 ilustra uma expansão do padrão MDA. Na parte superior, um modelo é preparado utilizando uma linguagem independente de plataforma especificada por um metamodelo. Na parte inferior, há um segundo modelo que utiliza uma linguagem de uma plataforma específica, especificada por outro metamodelo. Há uma transformação da primeira plataforma para a segunda. As especificações dessa transformação são em termos de mapeamento entre os metamodelos. Esse mapeamento orienta a transformação de PIM para produzir PSM. Além disso, esse processo possibilita uma transformação automatizada e pode ser empregada para traduzir modelos entre linguagens de mesmo domínio (OMG, 2003).

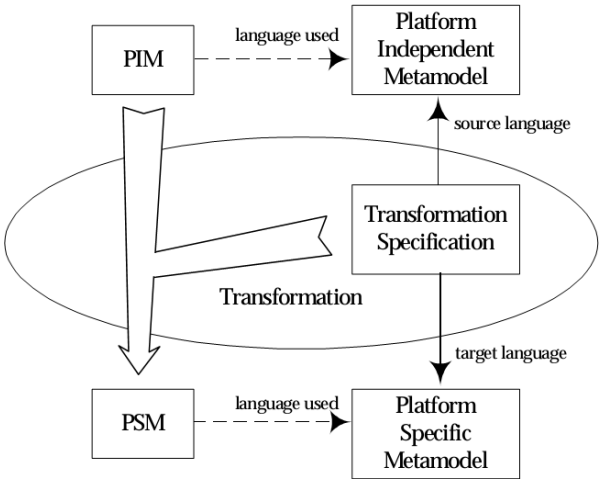


Figura 10: Transformação de Metamodelos (OMG, 2003)

Um metamodelo é um modelo que fornece base para construir outro modelo. Embora ambos sejam modelos, um é expresso em termos do outro, ou seja, um modelo é uma instância ou está em conformidade com o seu respectivo metamodelo (Gronback, 2009). Os metamodelos oferecem apoio a diferentes linguagens de modelagem, ajudam a garantir que modelos construídos estejam semanticamente

corretos e completos, possibilitam também a definição e execução de transformações. As principais abordagens de MDD são baseadas no conceito de metamodelagem (Lucrédio, 2009).

A arquitetura para metamodelagem pode ser dividida em quatro camadas ou metaníveis de modelagem (OMG, 2006): M3 - Metametamodelo, M2 - Metamodelo, M1 - Metadados e M0 - Objetos de Dados. A camada M3 - Metametamodelo define uma linguagem para descrever metamodelos. Não há uma camada acima desta, pois se considera que um metametamodelo é uma instância de si próprio. O *Meta Object Facility* (MOF) (OMG, 2006) consiste em um metamodelo que serve para definição de todas as linguagens de modelagem na MDA. Por exemplo, o *Ecore* é um exemplo de linguagem de metamodelagem utilizada pelo *Eclipse Modeling Framework* - EMF (Steinberg et al., 2008). Essa linguagem apoia a criação de metamodelos nesse framework. A camada M2 - Metamodelo define uma linguagem para criação de modelos. Por exemplo, a *Unified Modeling Language* (UML) é um exemplo de metamodelo que define elementos para a criação dos diagramas dessa linguagem. A camada M1 - Metadados ou Modelo especifica uma linguagem para descrever informações cobertas pelos domínios das aplicações. Por exemplo, um diagrama contendo as Classes “Estudante” e “Matricula” com seus respectivos atributos e tipos. Por fim, a camada M0 - Objetos de Dados compreende as instâncias dos modelos da camada M1 que correspondem aos dados usados na aplicação. Por exemplo: “*est = new Estudante(“Victor”)*”.

De modo geral, a MDA considera a modelagem no centro do processo de desenvolvimento de software. Assim vários modelos são utilizados para capturar diferentes aspectos do sistema de forma independente de plataforma e conjuntos de transformações podem ser aplicados nesses modelos independentes de plataforma, como transformações de modelo para modelo, modelo para texto (*Model to Text* – M2T) entre outros. Essencialmente o padrão M2T tem foco na geração de artefatos em modo texto a partir de modelos. Uma maneira intuitiva de atender esse requisito consiste em uma abordagem baseada em modelo no qual o texto a ser gerado é especificado por um conjunto de templates que são parametrizados com elementos do modelo (OMG, 2008).

Os templates especificam um modelo de texto com espaços reservados para os dados a serem extraídos a partir de modelos. Esses marcadores são essencialmente expressões especificadas sobre o metamodelo com consultas,

sendo os principais mecanismos para selecionar e extrair valores dos modelos. Esses valores são convertidos em fragmentos de texto, utilizando uma linguagem de especificação fornecida por uma biblioteca de manipulação de strings.

Por exemplo, a Figura 11 ilustra a especificação de um template para gerar classes Java a partir de um modelo UML. Na parte 1 da figura, apresenta-se um template, "[c.name/]" é um marcador para inserir o nome da classe proveniente do modelo. Na parte 2, tem-se um modelo UML contendo duas classes: "Employee" que estende "Person". Por fim, na parte 3, o código da classe Employee é gerado com a utilização do template.

Um gerador de código é um componente importante do MDD e para isso, há diversas ferramentas que além de um gerador, permitem a criação de templates. Dentre essas ferramentas, o *Java Emitter Template*[7] (JET) permite desenvolver templates, podendo ser usado na geração de SQL, XML, Java, Texto entre outros. Além do JET, o Acceleo também é um gerador de código, com uma linguagem que agrega como vantagem principal a facilidade para usuários inexperientes com esse tipo de tecnologia.

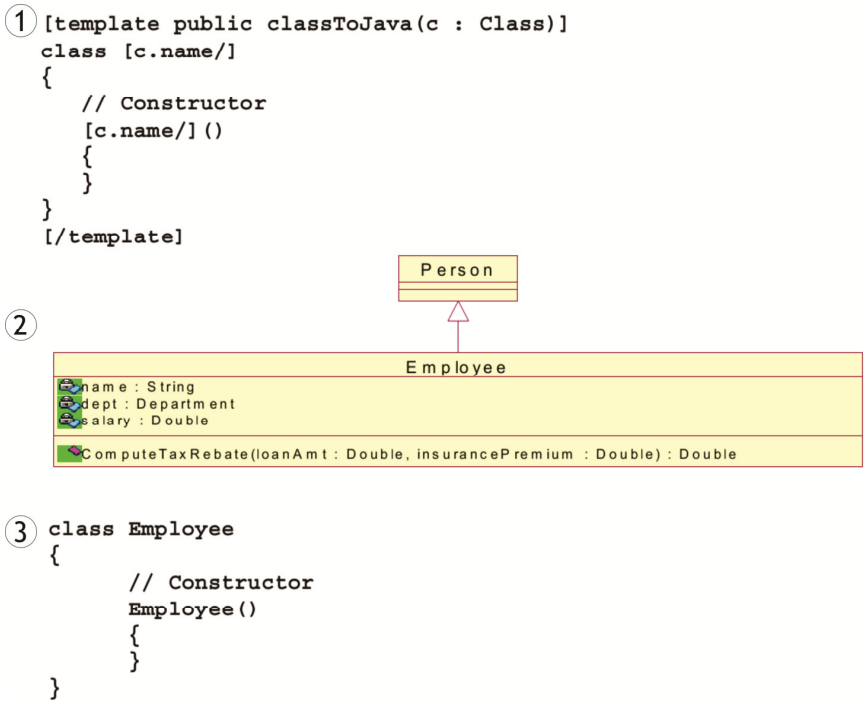


Figura 11: Exemplo da utilização de templates (adaptado de OMG, 2008)

[7] www.eclipse.org/emft/projects/jet/

Em síntese, as vantagens do MDD estão relacionadas à capacidade de evitar que o desenvolvedor tenha que executar tarefas repetitivas para a transformação de modelos para código, o que pode ser alcançado por meio da automação dessas transformações. O tempo despendido nessas tarefas, que no desenvolvimento convencional, desestimula a execução do ciclo completo dos requisitos aos testes, é significativamente reduzido graças às transformações automatizadas, o que faz com que mesmo atividades urgentes, como correção de erros, possam ser executadas sem deixar os modelos inconsistentes, mantendo-os atualizados constantemente.

3.2.1 Acceleo

Acceleo é um gerador de código a partir de modelos que segue o padrão *Model to Text Language* (MTL) do OMG. Esse gerador possui duas versões: a versão *open-source* que é distribuída sob a licença EPL[8] e uma versão comercial que possui suporte e treinamento especializado.

Essa ferramenta é resultado de vários anos de desenvolvimento da companhia francesa Obeo[9]. Segundo a equipe, o Acceleo foi criado considerando as necessidades do contexto industrial e recentes avanços de pesquisa no campo M2T, oferecendo vantagens como: alto nível de customização, interoperabilidade, facilidade para "ponto de partida", gerenciamento de rastreabilidade entre outras. O Acceleo está em constante aprimoramento e conta com manuais, fóruns, equipe de suporte e uma comunidade ativa. Em 2009, o Acceleo recebeu um prêmio como melhor ferramenta open-source baseada na plataforma Eclipse[10]. Desde 2009, o Acceleo é um projeto oficial da *Eclipse Foundation*, sendo uma referência na geração de código na plataforma Eclipse.

Com o Acceleo, pode-se gerar código Java além de outras linguagens de programação partindo de modelos. Essa geração é feita por meio de templates, nos quais podem ser inseridos *tags* que servem para rotular trechos do código que poderá ser gerado. Dessa maneira, consultas podem ser executadas no template, em que caso um trecho de código esteja rotulado com uma determinada *tag*, esse trecho poderá fazer parte ou não do código fonte gerado no final do processo. Vale

[8] http://www.eclipse.org/legal/epl-v10.html

[9] http://www.obeo.fr/

[10] http://www.obeo.fr/pages/open-source/en

ressaltar que tanto as consultas, quanto as *tags* são implementadas com a linguagem de especificação do Acceleo.

O Acceleo possui um editor de templates, contendo um assistente, realce de sintaxe, detecção de erros entre outros recursos. Além do gerador de código e o editor de templates, o Acceleo conta com um assistente para criação de *plug-ins*. Com esse artifício, o desenvolvedor pode, por exemplo, criar um *plug-in* para o Eclipse, especificando: (i) um metamodelo (um modelo *ecore* criado com apoio do EMF), (ii) a extensão das instâncias desse metamodelo que servirá de entrada para o Acceleo (no formato *xmi*), (iii) um conjunto de templates que devem ser considerados e (iv) um *parser* que transforme um modelo de outra extensão que se deseja utilizar, em uma instância do metamodelo a ser considerado. Caso o modelo de entrada seja um *xmi* e seu metamodelo esteja disponível, basta considerá-lo como entrada para o Acceleo. Por exemplo, no caso de geração de código a partir de modelos UML (em formato *xmi*), o Acceleo possui um metamodelo *ecore* correspondente. Assim, não há necessidade de desenvolver um *parser* nem um metamodelo, pois a partir de modelos UML o Acceleo é capaz de gerar código fonte.

O gerador de código do Acceleo, ao considerar os templates que contém as *tags* e um conjunto de consultas, o metamodelo e sua instância, torna-se capaz, em tempo de projeto, de gerar código fonte. Por exemplo, supondo a existência de um metamodelo “X” e uma instância “X.y”, cuja extensão criada foi “y” e um template principal T. O metamodelo X descreve um conjunto de elementos, com suas respectivas propriedades e relacionamentos. Cada elemento pode ser considerado como uma classe que pertence a um “esqueleto”. A instância “X.y” respeita a estrutura de “X” e contém dados que foram preenchidos. Se “X” tem uma Classe “A” com o atributo “a” do tipo String, a instância “X.y” pode ter vários elementos do tipo Classe “A” com o atributo “a” do tipo String contendo valores variados. Com base nessa instância, o Acceleo pode realizar uma análise no template T. Caso exista alguma Classe em “X.y” com a estrutura da Classe “A”, cujo atributo “a” contém o valor “zxc” e que exista também, um determinado trecho código no template T rotulado com a *tag* contendo uma condição correspondente, tal trecho poderá fazer parte do código gerado.

A Figura 12 ilustra a ideia geral de como o Acceleo transforma um modelo em código de forma convencional. Na Etapa 1, há um modelo que pode ser um diagrama de classes. Na Etapa 2, deve-se selecionar o metamodelo da linguagem

de programação, na qual pretende-se gerar código. Por exemplo: JEE, PHP, .NET e etc. A parte central da figura ilustra o mecanismo de geração de código do Acceleo. Considerando um metamodelo que contém a estrutura de uma linguagem de programação, considerando sua BNF[11], o gerador do Acceleo é capaz de gerar código na linguagem definida a partir do modelo, conforme ilustrado na Etapa 3.

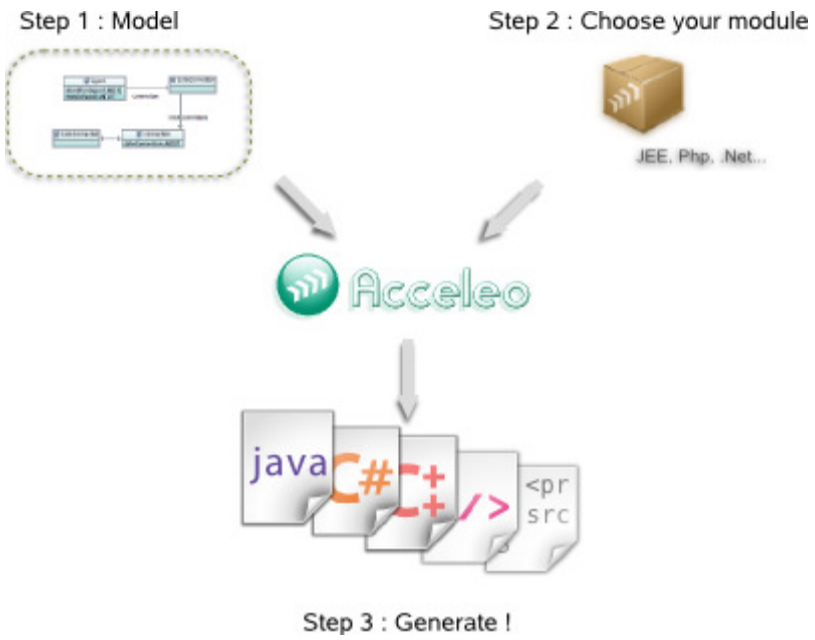


Figura 12: Visão geral do processo de geração de código com Acceleo (Acceleo, 2013)

Para exemplificar como os templates Acceleo podem ser implementados, na Figura 13 ilustra-se trechos de código de dois templates. Na parte (1) da figura, apresenta-se a implementação do template principal “main.mtl” (*mtl* é a extensão dos templates Acceleo). O template principal tem a função de realizar “chamadas” aos demais templates, caso necessário. Note-se que na parte superior há alguns *imports* a outros templates que são responsáveis pela geração de classes Java. Há também um laço “*for*” que serve para percorrer completamente o modelo fornecido como entrada para o Acceleo, ou seja, todos os elementos “feature” do tipo “Feature” que pertencem a “domain”. Vale enfatizar que essa estrutura é proveniente de um metamodelo. Em seguida, visualizam-se as *tag* condicionais do tipo “IF” no template principal. No caso, a parte interna do bloco “IF” está sendo utilizada para “chamar”

[11] BNF (*Backus Normal Form*) ou Formalismo de Backus-Naur é uma meta-sintaxe utilizada para expressar gramáticas livres de contexto, sendo ela a notação mais usual para descrever gramáticas de linguagens de programação (Naur, 1960).

os templates que geram as classes obrigatórias, ou seja, classes que pertencem ao "core" de um framework ou de uma LPS, por exemplo. Caso o modelo de entrada contiver um elemento feature com o atributo "*name*" com o valor "GRENJ", isso implica em gerar as classes: "AbstractCalculator", "QualifiableObject" e etc. Por exemplo, poderia ser `[if (feature.name.equalsIgnoreCase('X'))]` `feature.geradorClasseA()` `[/if]`, isso significa que se houver algum elemento do tipo feature com o atributo *name* com o valor X, indicando que a feature X foi selecionada, então o template responsável por gerar a "ClasseA" deve ser chamado.

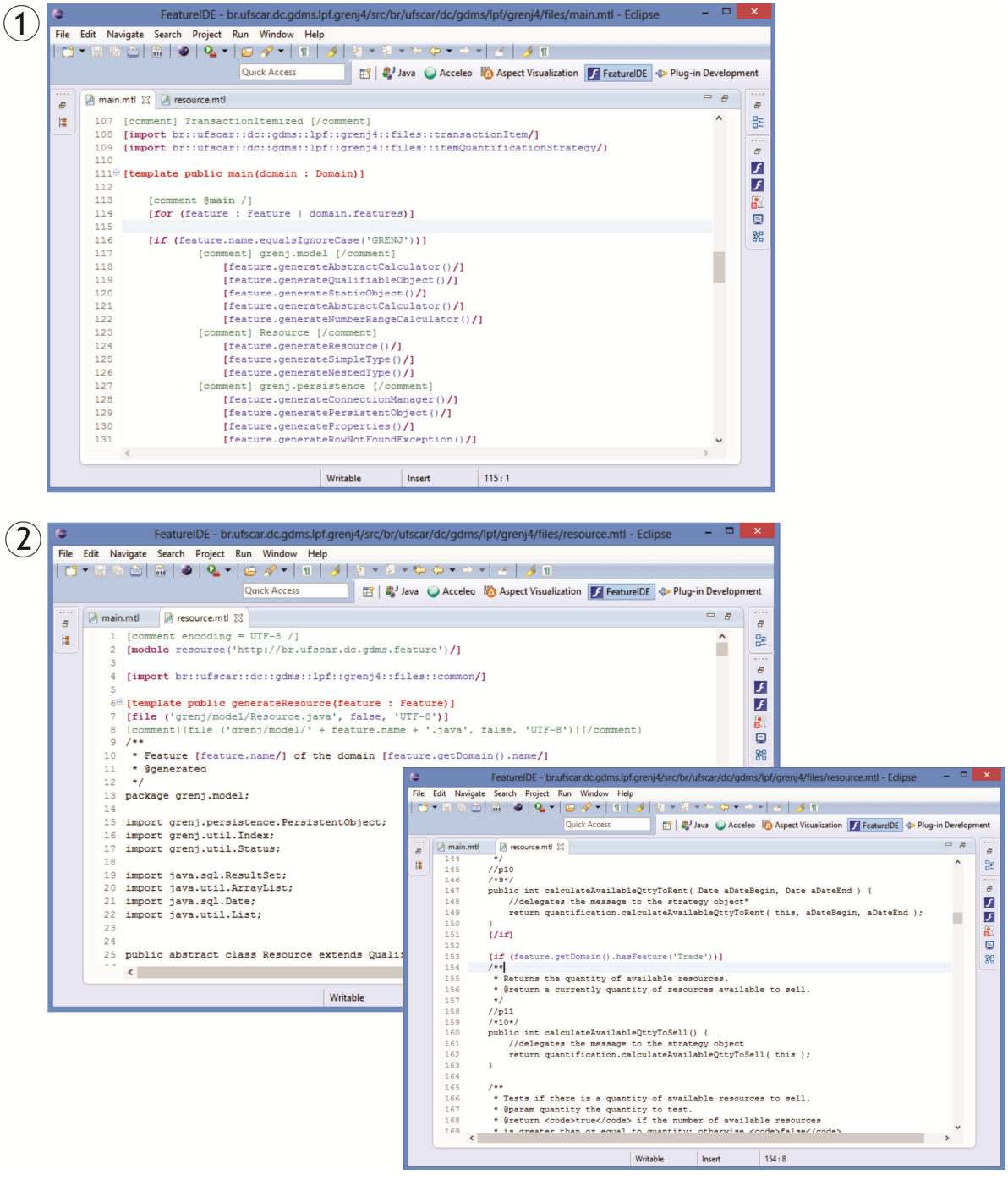


Figura 13: Exemplo de implementação de templates com Acceleo

Na parte (2) há um trecho do template *resource.mtl*. Esse template é responsável por gerar a classe "*Resource*". Note que na parte superior, em "module resource" é definido o metamodelo a ser considerado. Em seguida há um *import* de *common.mtl*, trata-se de um outro template que foi desenvolvido apenas para realização de consultas. Em seguida, têm-se alguns comentários e o código da Classe "Resource". No lado direito da parte (2) há outro trecho do template *resource.mtl* que contém um nível diferente de condições. No caso, **[if (feature.getDomain().hasFeature('Trade'))]** "trecho de código" **[/if]**, isso indica que se a feature "Trade*"* estiver no modelo de entrada, ou seja, há um elemento do tipo feature com o atributo *name* contendo "Trade", o trecho de código associado a essa *tag* deve ser adicionado ao código que será gerado. Observa-se que, na parte 1 o template principal é responsável por chamar todos os templates que geram as classes que implementam as features selecionadas. De outro lado, há features que impactam em outras features e que provavelmente afetam várias classes. Logo, torna-se necessário verificar quais features foram selecionadas, para que o código de uma classe que participe da implementação de diversas features, contenha somente os métodos e atributos das features selecionadas e que consequentemente pertencem ao modelo de entrada.

3.3 Considerações Finais

Os mecanismos fornecidos pela POA permitem a modularização de software evitando problemas como o entrelaçamento e espalhamento de código que dificultam a realização da manutenção, fundamental no ciclo de vida de um software. De outro lado, o MDD fornece meios para que um modelo seja transformado em código fonte considerando as melhores práticas/padrões de desenvolvimento. Além disso, o MDD permite ao desenvolvedor se concentrar em atividades mais importantes, considerando a possibilidade de evitar tarefas repetitivas que geralmente consomem maior parte do esforço no desenvolvimento.

Considerando tais abordagens, acredita-se que ambas podem ser aplicadas em frameworks amplos e com arquitetura rígida, seja com o intuito de melhorar sua modularidade ou solucionar dificuldades inerentes à quantidade de variabilidades

desnecessárias para determinados conjuntos de aplicações desenvolvidas com seu apoio. No capítulo seguinte, apresenta-se uma discussão acerca dessas dificuldades e uma alternativa para suavizá-las. Essa alternativa envolve a modularização das variabilidades do framework em features, possibilitando alcançar níveis mais efetivos quanto à flexibilidade de composição. Assim, podem-se compor frameworks direcionados às preferências do usuário.

Capítulo 4

FRAMEWORKS DE MÚLTIPLOS DOMÍNIOS E LINHA DE PRODUTOS DE FRAMEWORKS

Neste capítulo discute-se com mais profundidade os problemas que motivam a modularização de Frameworks de Múltiplos Domínios nas chamadas Linha de Produtos de Frameworks. Assim, abordamos as principais características dos FMDs, bem como os refinamentos dos conceitos e mecanismos das Linhas de Produtos de Frameworks para tratar tais limitações.

4.1 Frameworks de Múltiplos Domínios

Uma das principais características dos frameworks é que são específicos de domínio, isto é, oferecem abstrações de um domínio (Johnson, 1991; Gamma, et al. 1995). Assim, naturalmente um framework é considerado satisfatório quando suas variabilidades são potencialmente prováveis de serem usadas nas aplicações do domínio ao qual ele oferece suporte. Na maioria dos casos, não há problema se apenas uma pequena porcentagem das features é usada por uma aplicação, desde que exista a possibilidade das features remanescentes serem utilizadas futuramente. Como as aplicações podem evoluir com o tempo, as demais features podem ainda ser necessárias.

Em geral, frameworks se tornam amplamente empregados e bem aceitos quando oferecem uma enorme gama de features que atendem as mais diversificadas necessidades de implementação. Em consequência disso, é natural

que os frameworks evoluam nessa direção, sempre agregando novas variabilidades para satisfazer as necessidades de seus potenciais usuários. Quando essas evoluções são realizadas sem um gerenciamento ou planejamento adequado, as variabilidades incorporadas podem extrapolar o domínio previamente planejado para o framework, resultando em "Framework de Múltiplos Domínios (FMD)".

Em alguns casos, ao desenvolver uma aplicação com o apoio de um FMD caixa branca, pode ser necessário concretizar métodos abstratos referentes a variabilidades não utilizadas pela aplicação e isso é algo que não deveria ocorrer. Caso o FMD seja caixa preta, outro problema é a presença de muitas variabilidades desnecessárias durante o processo de reuso, prejudicando a produtividade e podendo implicar em erros. Por exemplo, ao utilizar o framework Hibernate em algum ambiente de programação, uma maneira de selecionar as variabilidades consiste em digitar por um período e selecionar uma das variabilidades em uma lista. Caso o desenvolvedor esteja desenvolvendo uma simples aplicação, que usa poucas variabilidades, ou uma grande e complexa, que usa muitas, o mesmo conjunto de variabilidades aparecem nessa lista. Dessa maneira, considerando um desenvolvedor que atua em um domínio restrito, ele está sujeito a conviver com uma série de variabilidades que nunca irá utilizar. Quando todas as variabilidades estão disponíveis para os desenvolvedores, pode levar maior tempo para selecionar somente as apropriadas ou até mesmo, permitir a seleção acidental. Essas ocorrências também podem contribuir com uma maior tendência a erros no processo de instanciação.

De modo geral, admite-se que os FMDs sejam resultantes de evoluções realizadas de forma indiscriminada para apoiar o desenvolvimento de aplicações que possuem subdomínios distintos. Porém, cada aplicação desenvolvida com o apoio de um FMD, o incorpora integralmente, mesmo que utilize ou futuramente seja utilizada apenas a parte referente a um subdomínio. Possivelmente, a arquitetura desses FMDs pode ser rígida em termos de composição e decomposição de suas variabilidades, fator que impossibilita criar frameworks menores para atender demandas específicas para essas aplicações.

Uma forma de identificar se um framework é um FMD é mediante a análise de um conjunto de aplicações desenvolvidas com seu apoio ou até mesmo o fato de conhecer completamente a estrutura do framework pode ser suficiente. A Figura 14 ilustra a ideia geral das características desse tipo de framework. Na parte (I) da

Figura há dois conjuntos. O primeiro conjunto, chamado de F, representa um framework com todas suas 9 variabilidades. O segundo conjunto, denominado pela letra A, representa um repositório contendo 17 aplicações desenvolvidas com o apoio de F. Cada variabilidade é representada por uma "peça de quebra-cabeça" contendo o prefixo "v" seguido por um valor numérico, enquanto que as aplicações são representadas por um quadrado contendo o prefixo "a" seguido também por um valor numérico.

Supondo que existe um artefato de mapeamento entre as aplicações e as variabilidades de F, o qual permite identificar quais variabilidades cada aplicação está utilizando, podem-se criar subconjuntos com as variabilidades de F. Conforme ilustrado na parte (II) da Figura, foram criados os subconjuntos S_1, S_2 e S_3. Para a formação desses subconjuntos, foi considerada a frequência com que as variabilidades foram utilizadas simultaneamente nas aplicações. Além disso, para criar tais subconjuntos, foi necessário averiguar se as variabilidades contidas são suficientes para apoiar não somente o desenvolvimento, como também a evolução dessas aplicações. Por exemplo, a aplicação a_4 utiliza somente as variabilidades v_4 e v_6 e futuramente pode ser evoluída, utilizando também as variabilidades v_1 e v_9, mas dificilmente utilizará as variabilidades que estão nos subconjuntos S_2 e S_3.

Vale ressaltar que podem existir aplicações que abrangem mais de um desses subconjuntos. Entretanto, a ideia dessa separação consiste em abstrair as variabilidades para determinados grupos de aplicações que utilizam ou futuramente utilizarão somente as variabilidades restritas nesses subconjuntos.

Nota-se também que existem variabilidades comuns aos subconjuntos formados, como é o caso da variabilidade v_4 que é utilizada por todas as aplicações analisadas. Assim, essa variabilidade pertence a todos os subconjuntos.

No caso do framework F, podem-se abstrair três subdomínios a partir dos subconjuntos S_1, S_2 e S_3. Diante disso, o ideal seria a existência de três frameworks menores, um para cada subdomínio. Porém, considerando a forma como o framework é reutilizado, mesmo que existam aplicações que se limitem à utilização de somente um conjunto menor de variabilidades disponibilizadas por F, todas as variabilidades estarão presentes nas 17 aplicações. Esse fato indica que F possui características de um FMD.

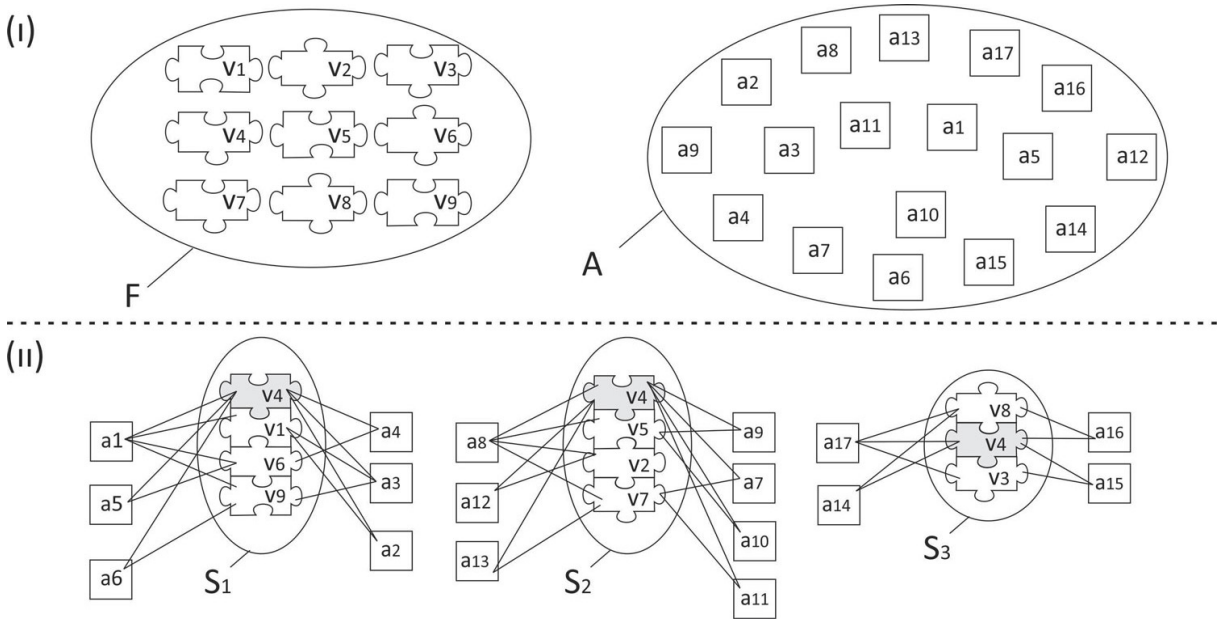


Figura 14: Framework de Múltiplos Domínios

De modo geral, a problemática de um FMD está relacionada com os seguintes pontos: (i) arquitetura inflexível, o que impossibilita gerar frameworks menores para determinados conjuntos de aplicações, refletindo na presença de features desnecessárias do framework na arquitetura final das aplicações, (ii) as variabilidades opcionais do framework não são expressas de forma coerente, pois se fossem, ao reutilizar um framework e a aplicação não exigir uma determinada variabilidade que é opcional, o desenvolvedor deveria ter a opção de não incluí-la no *release* final da aplicação, (iii) presença de muitas funcionalidades que não possuem um papel específico no framework, pois colaboram com a implementação de vários subdomínios, (iv) processo de reúso com alta tendência a erros, pois em alguns casos pode ser necessário concretizar métodos de variabilidades desnecessárias à aplicação e (v) dificuldade para expor as funcionalidades disponibilizadas pelo framework de maneira clara para o usuário do framework.

Considerando essa problemática, torna-se evidente a necessidade de evoluir esse tipo de framework, no sentido de tratar suas limitações. Dessa maneira, apresenta-se na Seção 4.2 o conceito das “Linhas de Produtos de Frameworks”. Essa abordagem, além de ser independente de uma plataforma e linguagem específica de implementação, permite ao EF contornar as dificuldades mencionadas anteriormente.

4.2 Linha de Produtos de Frameworks

Considera-se uma “Linha de Produtos de Frameworks” (LPF), uma linha de produtos de software cujos membros são frameworks ao invés de aplicações de software completas. A principal diferença entre LPS e LPF está relacionada com seus membros; a composição de features de uma LPS resulta em aplicações de software funcionais, enquanto que em uma LPF resulta em frameworks que ainda precisam ser instanciados ou acoplados a um código base para obter aplicações concretas.

A Figura 15 ilustra a ideia principal de uma LPF. Na parte (a), dois modelos de features representam versões distintas da mesma LPF. Na parte (b) existem alguns frameworks gerados a partir dessas versões. Na parte (c), podem-se observar aplicações geradas a partir desses frameworks.

Na parte (a), a diferença entre as duas versões da LPF está relacionada com a granularidade das features. A primeira, na parte esquerda, possui além da feature *Core*, outras três features que representam os subdomínios. A segunda versão, na parte direita, possui mais features em relação à primeira, mas possui o mesmo número de variabilidades, a diferença é que suas features estão em nível mais baixo de abstração.

Na parte (b), estão os “membros” que podem ser derivados a partir das LPFs. Na primeira, somente três combinações são possíveis. Na segunda, existem mais possibilidades e o usuário pode compor um framework segundo os requisitos das aplicações ao invés de considerar apenas o subdomínio.

Na parte (c), existem seis aplicações que foram desenvolvidas com membros das duas versões da LPF. Cada aplicação é representada por duas parcelas: framework e código base da aplicação. Nota-se que o framework 2 possui o mesmo tamanho, em termos de variabilidades, que o framework 5, mas isso não acontece entre os frameworks 1/4 e 3/6. Isso demonstra que para as aplicações 1,2,5 e 6, versões reduzidas do framework podem ser suficientes. A questão é que se essas aplicações evoluírem e for necessária nova feature que não esteja nas versões utilizadas, será necessário buscá-las na LPF e em seguida agregar ao framework. No caso dos frameworks provenientes da LPF da esquerda, podem existir

variabilidade remanescentes, porém como fazem parte do subdomínio da aplicação, há possibilidade de serem usadas no futuro.

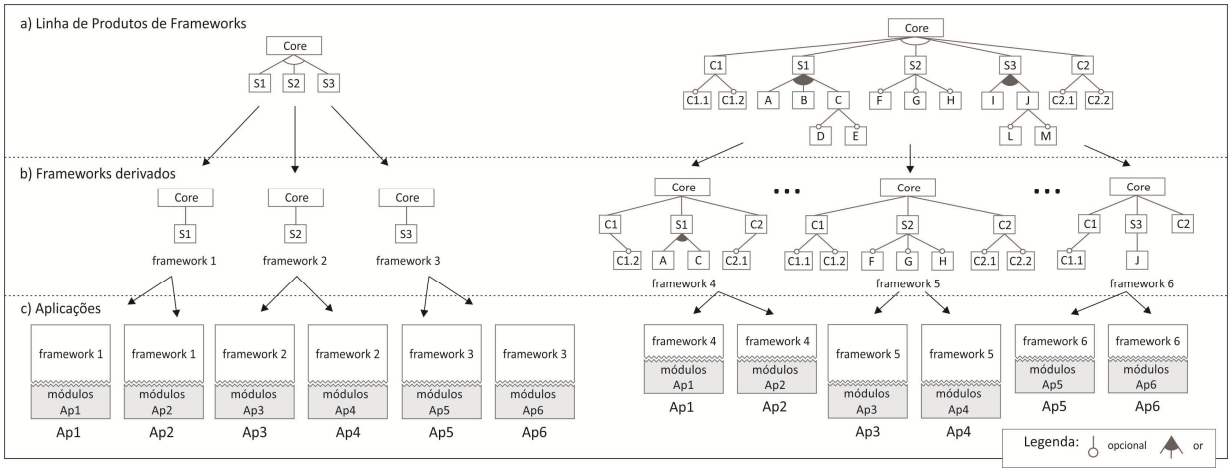


Figura 15: Linha de Produtos de Frameworks

4.2.1 Cenários de Utilização

Considerando os conceitos apresentados anteriormente, há dois cenários de utilização para uma LPF. Esses cenários podem ser considerados como o principal critério para determinar a granularidade das features. No **Cenário de Utilização 1**, uma LPF possibilita a geração de poucos, mas amplos frameworks, ou seja, contendo um pequeno conjunto de features que encapsulam todas as variabilidades existentes. No **Cenário de Utilização 2**, consideramos a existência de uma LPF que permite a criação de amplos frameworks, como também a criação de frameworks menores que fornecem um conjunto limitado de features direcionado aos requisitos específicos das aplicações.

Com relação à granularidade das features: no Cenário de Utilização 1, a LPF possui features de granularidade grossa, ou seja, existem muitas variabilidades agrupadas em poucas features e com isso, há um número reduzido de combinações. Como etapa opcional, essa LPF pode ainda ser modularizada no sentido de obter features mais granulares. Essa etapa representa o Cenário de Utilização 2, em que uma LPF possui features de granularidade fina, isto é, existem muitas features com diversas alternativas de composição. Para a modularização de um FMD de acordo com o Cenário de Utilização 2 deve-se primeiro modularizar as features em

granularidade grossa e, posteriormente, decompõe-se essas features em nível mais granular.

O principal ponto nesses cenários de uso é que uma LPF deve ser modularizada de forma a não permitir a geração de frameworks com features que nunca serão utilizadas por determinados conjuntos de aplicações (pertencentes a um domínio). Acredita-se que a situação mais comum seja a existência de uma LPF com poucas features para a geração de amplos frameworks, conforme indicado pelo Cenário 1. Entretanto, caso necessário, pode-se construir uma LPF com um número maior de features de granularidade fina visando a geração de vários frameworks restritos, tal como indicado pelo Cenário 2. Vale ressaltar que no Cenário 2, quando as aplicações evoluem, podem requerer funcionalidades que não existem na versão restrita do framework utilizado. Assim, nesse cenário é indispensável uma estratégia para seleção e composição de features sob demanda. Portanto, deve existir um mecanismo para buscar novas features na LPF e, encontrando-as, realizar a composição com o framework.

No caso de uma LPF obtida segundo o Cenário de Utilização 1, existem features de granularidade grossa que podem ser referentes aos subdomínios. A partir dessa LPF, as features podem ser decompostas em nível de granularidade fina, expondo as variabilidades inclusas nesses subdomínios, conforme apontado no Cenário de Utilização 2. Assim, a motivação para o Cenário de Utilização 2 consiste em, além de identificar e modularizar os subdomínios, permitir que um conjunto de features mais restrito seja gerado. Esse conjunto refere-se a uma parcela das variabilidades cobertas por um subdomínio e/ou das que foram consideradas comuns aos subdomínios na LPF obtida com base no Cenário de Utilização 1. Possivelmente, essas features comuns foram consideradas obrigatórias na LPF referente ao Cenário de Utilização 1, mas na LPF segundo o Cenário de Utilização 2, parte delas podem se tornar opcionais.

Um conjunto de features de uma LPF conforme o Cenário de Utilização 1, pode não ser utilizado durante a instanciação; no entanto, uma vez que as features utilizadas por essas aplicações pertencem ao mesmo subdomínio, é provável que sejam utilizadas futuramente. Isso significa que as aplicações podem evoluir sem correr o risco de ter que buscar features que não estejam no framework. No entanto, o número de features que são levadas nas aplicações é maior do que um framework derivado de uma LPF segundo o Cenário 2.

Vale ressaltar que quanto mais granulares as features de uma LPF, maior o esforço para modularizá-las, pois isso envolve o tratamento de interferências e dependências entre as features. O número de pontos para ser tratado no código é maior em comparação com as features que envolvem grandes blocos, como as features que contém todas as variabilidades dos subdomínios.

4.2.2 Features de uma LPF

Considerando que os frameworks derivados de uma LPF podem ser do tipo caixa branca, preta ou cinza, há alguns pontos que devem ser enfatizados. Caso o objetivo seja gerar frameworks caixa branca, a qualidade do código do membro em termos de instanciação é essencial para o EA, uma vez que ele deverá instanciar o framework e precisará observar diretamente o código. Assim, a tecnologia/técnica utilizada na construção de uma LPF não pode comprometer a qualidade do código. Por exemplo, a utilização de diretivas de compilação condicional `#ifdef` pode "ofuscar" o código fonte (Loesch e Ploedereder, 2007; Kästner et. al, 2008). Caso o objetivo seja gerar frameworks caixa preta, o membro pode ser compilado ou ser um "`.jar`", logo as possibilidades de instanciação são previamente disponibilizadas. Desse modo, a qualidade do código é mais importante em termos de manutenção, uma vez que isso ficará sob responsabilidade do EF.

Para o Engenheiro de uma LPF, o subdomínio das aplicações que se deseja atender é o fator que pode direcionar a seleção das features para a composição de um membro. Assim, quando se define um subdomínio, faz sentido levar todas as features inclusas nesse subdomínio para diversas aplicações, afinal o membro é um framework. Em uma LPS convencional, isso seria diferente, pois a seleção de features é dirigida essencialmente segundo as preferências de seu usuário, ou seja, consideram-se os requisitos específicos de uma aplicação que se deseja construir, logo não é coerente a pertinência de funcionalidades cobertas pela LPS que não fazem parte do escopo da aplicação.

Essa diferença entre LPS e LPF com relação à seleção de features tem influência na forma como elas devem ser organizadas em um diagrama de features, principalmente quanto à associação "*or*" e da relação alternativa (xor) que é mutuamente exclusiva. Por exemplo, na análise de um subconjunto de features é comum notar que algumas delas podem ser agrupadas a uma categoria, ou seja,

são subfeatures de uma feature que está em um nível superior no modelo. Diante disso, deve-se decidir: utilizar a associação por meio do conector "*or*" ou optar pela relação alternativa (*xor*). É importante ressaltar que por meio do conector "or" permite-se que todas as subfeatures possam fazer parte do membro da LPF e com a relação "*xor*", apenas uma delas. Em muitos casos, decidir pelo relacionamento "*or*" em uma LPF pode ser mais coerente, por permitir que todas as features pertencentes a um determinado grupo de variabilidades pertençam ao framework. Todavia, se essa análise fosse realizada sobre o ponto de vista de uma LPS, na qual o objetivo consiste em gerar aplicações executáveis, um relacionamento alternativo (xor) poderia ser mais adequado, pois dependendo do domínio não faria sentido uma aplicação com todas as variabilidades da mesma categoria e a aplicação requerer somente uma delas.

4.2.3 Desenvolvimento de uma LPF

A experiência relatada neste trabalho diz respeito ao desenvolvimento de uma LPF a partir da modularização de um único framework, no entanto uma LPF pode ser desenvolvida a partir da análise de um domínio ou vários frameworks que cobrem o mesmo domínio.

Uma LPS pode ser extraída a partir de diversas aplicações e envolve o desenvolvimento dos chamados "*core assets*" e produtos que utilizam esses *assets* (Clements e Northrop, 2002), enquanto que uma LPF pode ser obtida com análise de vários frameworks que apresentam funcionalidades específicas e comuns, considerando a utilização dessas features por certo número de aplicações.

Para o desenvolvimento de uma LPF, além de analisar as aplicações desenvolvidas com o apoio de um ou mais frameworks, deve-se definir o cenário de utilização que a LPF deve satisfazer, conforme descrito anteriormente. Em seguida, a partir da análise das features utilizadas por um ou mais frameworks, pode-se identificar as features, seus relacionamentos e como podem ser organizadas. Assim, um diagrama de features com as possíveis combinações pode ser criado. Depois disso, é necessário modularizar o framework considerando essas features. Provavelmente, em nível de código, as features estarão entrelaçadas e espalhadas em vários subdomínios, ocorrência comum no desenvolvimento orientado a objetos. Com base no diagrama de features criado, parte-se para a modularização, na qual

se aplica uma técnica para decompor o framework de forma a considerar o modelo de features e consequentemente, também as demandas que a LPF obtida no final do processo deve atender. Os detalhes desse processo de modularização são apresentados no Capítulo 5.

4.3 Considerações Finais

Conforme discutido neste capítulo, os problemas que envolvem os FMDs implicam tanto na manutenção quanto no processo de reúso. Como alternativa para suavizar tais limitações, tem-se a modularização desses frameworks em LPFs. Essas LPFs possibilitam gerar frameworks direcionados aos domínios/requisitos das aplicações, evitando que features que dificilmente serão utilizadas, permaneçam no *release* final das aplicações.

Com base nos conceitos a cerca das LPFs, partimos para a apresentação das diretrizes que compõem um processo de modularização de um FMD para uma LPF. Nesse processo, os conceitos das LPFs e a questão dos Cenários de Utilização são considerados.

Capítulo 5

FMDTOLPF – UM PROCESSO DE MODULARIZAÇÃO DE FMD PARA LPFS

Um processo de modularização de FMD para LPF é apresentado neste capítulo. O processo é denominado "FMDtoLPF" e possui um conjunto de diretrizes, considerando os refinamentos dos conceitos de LPFs discutidos previamente. Essas diretrizes são apresentadas de forma sequencial e podem ser empregadas na modularização de outros frameworks.

5.1 Visão Geral do Processo de Modularização

Para a concepção do processo *FMDtoLPF* foi necessário conduzir um estudo de caso para obter uma LPF a partir de um FMD e a partir disso, extrair um conjunto de diretrizes que possa ser aplicado na modularização de outros frameworks. Para isso, foi conduzido um estudo de caso utilizando o framework GRENJ (Durelli et al., 2010), desenvolvido com base em uma linguagem de padrões chamada GRN (Braga et al., 1999) que cobre o domínio de Gerenciamento de Recursos de Negócios. O GRENJ contém 8,5 *KLOC*[12] sendo composto por três pacotes, 71 unidades de implementação (classes/interfaces) e 1117 métodos. Esse framework tem sido estudado em nosso grupo de pesquisa há um longo tempo e muitas aplicações hipotéticas têm sido derivadas dele. Vale ressaltar que o processo é independente

[12] *Kilo Lines of Code* (mil linhas de código), é uma medida para tamanho de grandes programas de computadores.

de linguagem/plataforma específica e deve ser aplicado manualmente, pois não há apoio automático.

Na Figura 16 ilustra-se uma visão geral das duas grandes fases do processo. Essas fases possuem um alto nível de abstração, uma vez que incluem várias atividades que posteriormente são detalhadas.

Figura 16: Visão Geral do processo *FMDtoLPF*

A fase de *Análise* é caracterizada pela identificação de subdomínios disjuntos e definição das features que devem ser modularizadas na fase seguinte. Para isso, deve-se considerar toda informação disponível sobre o framework. Caso possível, sugere-se averiguar aplicações desenvolvidas com o apoio do framework, no sentido de identificar variabilidades que sejam mais utilizadas por determinados grupos de aplicações e as que não sejam usadas por outros. Como artefatos dessa fase, são elaborados: um *Diagrama de features*, um artefato que descreve a *Ordem de Prioridade para Análise/Modularização dos Subdomínios* e um *Mapeamento entre unidades de implementação e as features*. Nessa fase, também se define o *Cenário de Utilização* que a LPF obtida no final do processo deverá atender. Caso os membros da LPF precisem ser restritos, isto é, com poucas variabilidades e que podem não apoiar o desenvolvimento de muitas aplicações, admite-se que essa LPF segue os princípios do Cenário de Utilização 2. Caso contrário, se cada membro deve fornecer todas as variabilidades de um determinado subdomínio, admite-se que a LPF leva em conta o Cenário de Utilização 1, conforme discutido no Capítulo 4.

Na fase de Modularização de features, deve-se optar por uma tecnologia que possa ser empregada na modularização das features identificadas, apesar do processo e dos conceitos serem independentes de uma tecnologia específica. Nessa definição, é importante atentar-se ao esforço necessário na decomposição/composição das features e como a manutenção pode ser realizada. Com essa definição, pode-se modularizar as features com base nos artefatos gerados na fase anterior. Essa fase é concluída quando todas as features estão

modularizadas, constituindo uma LPF que permite compor membros conforme o Cenário designado e respeitando o *Diagrama de Features*.

Com base nessa visão geral, na Figura 17 ilustram-se as fases em nível mais baixo de abstração, isto é, as atividades internas que constituem ambas as fases do processo. Essas atividades são apresentadas no formato SADT (*Structured Analysis and Design Technique*) (Ross, 1977) e estão divididas nas duas fases descritas anteriormente por uma linha tracejada. Cada atividade possui um valor na extremidade superior esquerda que serve para referenciá-las e indicar a sequência do processo.

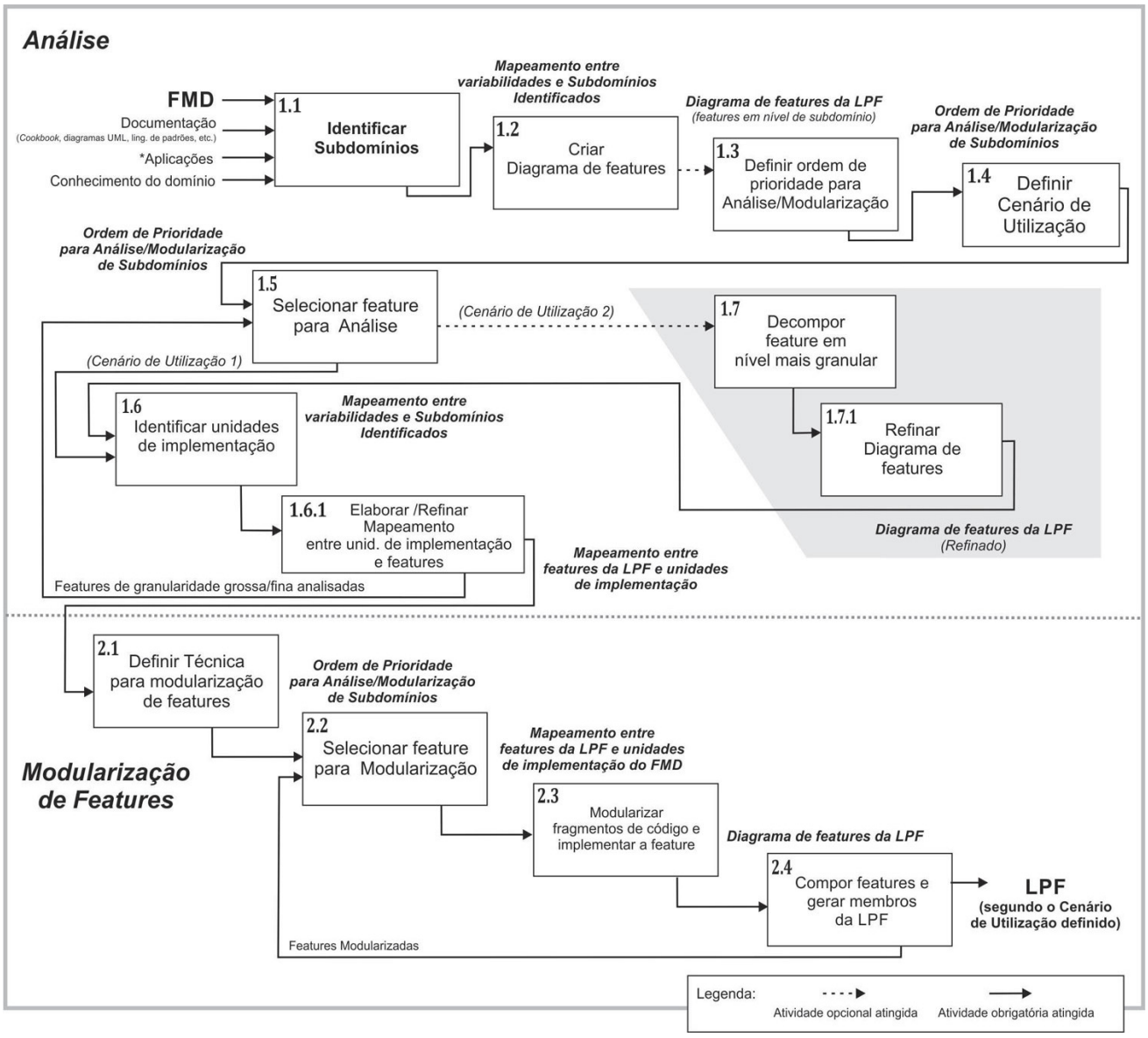


Figura 17: Visão detalhada do processo *FMDtoLPF*

Na atividade 1.1 - **Identificar Subdomínios**, toda a informação disponível do framework deve ser considerada para identificação dos subdomínios, como documentação, conhecimento prévio, aplicações desenvolvidas com seu apoio, etc. Com a existência dessas aplicações, devem-se averiguar quais variabilidades são mais utilizadas e aquelas que apesar de não estar sendo, são possíveis futuramente. Dessa maneira, podem-se definir determinados grupos de aplicações que indicam a existência de subdomínios disjuntos. Com isso, deve-se elaborar um artefato de mapeamento entre as variabilidades do framework e os subdomínios identificados.

Com base nesse mapeamento, parte-se para atividade 1.2 - **Criar Diagrama de features**, na qual um diagrama de features de alto nível deve ser elaborado para representar os subdomínios, levando em conta combinações válidas e restrições. Em seguida, parte-se opcionalmente para a atividade 1.3 - **Definir ordem de prioridade para Análise/Modularização**, que é caracterizada pela criação de um artefato para indicar qual subdomínio deve ser analisado e modularizado primeiro e qual a ordem de seus sucessores. Essa definição pode envolver dependências entre os subdomínios ou necessidades organizacionais.

Na atividade 1.4 - **Definir Cenário de Utilização** deve-se optar pelo Cenário de Utilização que a LPF que será obtida no final do processo deve satisfazer. Esses Cenários são discutidos no capítulo 4. Após essa definição, parte-se para a atividade 1.5 - **Selecionar feature para Análise**, na qual o subdomínio de maior prioridade presente no artefato gerado na atividade 1.3 deve ser selecionado para uma iteração de análise mais detalhada.

Caso a motivação seja obter uma LPF segundo o Cenário de Utilização 1, parte-se para a atividade 1.6 - **Identificar unidades de implementação**. Considerando que a feature selecionada refere-se a um subdomínio, essa análise detalhada deve ser realizada nesse nível. Com isso, elabora-se um artefato de mapeamento entre as unidades de implementação analisadas com as respectivas features, conforme descrito na atividade 1.6.1 - **Elaborar/Refinar Mapeamento entre unidades de implementação e features**. Caso a motivação seja obter uma LPF conforme o Cenário de Utilização 2, a partir da atividade 1.5, prossegue-se para a atividade 1.7 - **Decompor feature em nível mais granular**, na qual deve-se investigar quais variabilidades pertencentes ao subdomínio selecionado, podem ser dissociadas como novas features. Essa investigação é realizada em nível funcional

assim como a atividade 1.1, para que após definição de uma nova feature, seja possível analisar o código e identificar quais unidades colaboram com sua implementação.

Com a identificação dessas features de granularidade fina, parte-se para a atividade 1.7.1 - **Refinar Diagrama de features**. Essa atividade é caracterizada pela adição das features dissociadas dos subdomínios e da feature *Core* ao diagrama elaborado na atividade 1.2. Posteriormente, parte-se para a atividade 1.6 e consequentemente para a atividade 1.6.1, com a diferença que as features consideradas não correspondem somente aos subdomínios, como também as features que foram dissociadas no diagrama. Vale destacar que existe a possibilidade de obter uma LPF segundo o Cenário de Utilização 2, elaborando primeiro, o artefato de mapeamento entre as features e as unidades de implementação levando em conta somente os subdomínios e o *Core*, e depois em outra iteração, seguir no caminho à direita da atividade 1.5 e refinar esse artefato com as novas features que foram externalizadas.

Nota-se que a atividade 1.6.1 é a última atividade da fase de Análise e o requisito para partir para a atividade 2.1 - **Definir Técnica para modularização de features** é que todas as features tenham sido analisadas e mapeadas com as unidades de implementação disponíveis. Caso contrário, deve-se voltar para a atividade 1.5 e repetir o processo.

Na atividade 2.1, deve-se decidir qual a técnica mais adequada para modularizar as features. Com essa definição, parte-se para a atividade 2.2 - **Selecionar feature para Modularização**, que é caracterizada pela seleção de uma feature que deverá ser modularizada. As features selecionáveis na atividade 2.2 poderão ser as referentes aos subdomínios e ao *Core*, como também podem ser as que foram externalizadas na atividade 1.7. Após essa escolha, parte-se para a atividade 2.3 - **Modularizar fragmentos de código e implementar a feature**. Nessa atividade, deve-se modularizar os fragmentos de código das unidades que colaboram com a implementação da feature selecionada. Isso pode ser feito mediante o mapeamento entre features da LPF e unidades de implementação, gerado na atividade 1.6.1.

Após essa atividade, parte-se para a atividade 2.4 - **Compor features e gerar membros da LPF**, na qual as features que estiverem modularizadas podem ser compostas, gerando membros da LPF conforme o diagrama de features criado na

atividade 1.2 ou refinado na atividade 1.7.1. Dessa forma, como as combinações possíveis e restrições entre as features foram anteriormente analisadas, obtêm-se membros válidos. Em seguida, deve-se voltar à atividade 2.2 até que todas as features estejam modularizadas, constituindo uma LPF segundo o Cenário de Utilização definido no início do processo.

Nas subseções seguintes, as atividades do processo *FMDtoLPF* mostradas na Figura 17 são detalhadas. Com o intuito de torná-las mais inteligíveis, as mesmas são exemplificadas gradativamente em um estudo de caso que envolve a modularização do GRENJ até obter uma LPF.

5.2 Fase de Análise

A fase de Análise concentra todas as atividades em que o EF deve compreender completamente as funcionalidades presentes nos subdomínios cobertos pelo FMD. Para isso, além de identificá-los, é fundamental descobrir as dependências entre eles e os relacionamentos com a parte obrigatória do framework. Nessa fase, pode-se definir uma ordem de prioridade para análise e modularização dos subdomínios. Além disso, é importante estabelecer qual dos Cenários de Utilização, discutidos anteriormente, é o mais adequado para a LPF que se deseja obter, uma vez que isso implica na granularidade das features e consequentemente no esforço para modularizá-las.

5.2.1 Identificar Subdomínios

O primeiro passo do processo *FMDtoLPF* ilustrado na Figura 17 pela atividade 1.1 - **Identificar Subdomínios**, consiste em identificar se o framework trata de subdomínios disjuntos. Para isso, toda a informação disponível do framework pode ser considerada, como *Cookbook*, linguagem de padrões, conhecimento prévio e aplicações desenvolvidas com apoio do framework. Essas informações podem contribuir com maiores esclarecimentos sobre a arquitetura do framework, no sentido da identificação dos módulos principais, variabilidades disponíveis e para quais tipos de aplicações são destinadas.

Caso existam aplicações disponíveis que foram desenvolvidas com o apoio do framework, pode-se realizar uma análise que segue a ideia anteriormente apresentada na Figura 14. Para tornar mais clara essa atividade, considera-se inicialmente dois conjuntos, conforme ilustrado na parte (I) da Figura 18. O conjunto "F" contém todas as 9 variabilidades de um framework. O conjunto "A" contém 17 aplicações que foram desenvolvidas com o apoio desse framework. As variabilidades do framework são representadas por "peças de um quebra-cabeça" que contém um prefixo "v" seguido por um valor numérico, enquanto que as aplicações são representadas por quadrados contendo um prefixo "a" seguido também por um valor numérico. Essas variabilidades são comumente tratadas como variações nas classes abstratas dos frameworks (Batory et al., 2000). Segundo Parsons et al. (1999) e Johnson e Foote (1988), as variabilidades de um framework referem-se aos pontos variáveis (*hot spots*) que podem ser adaptados às necessidades de uma aplicação. Nesse trabalho, o termo variabilidade no contexto de frameworks, é entendido como uma funcionalidade específica que possui variações (Coplien et al., 1998), isto é, escolhas diferentes que dependem dos requisitos de uma determinada aplicação.

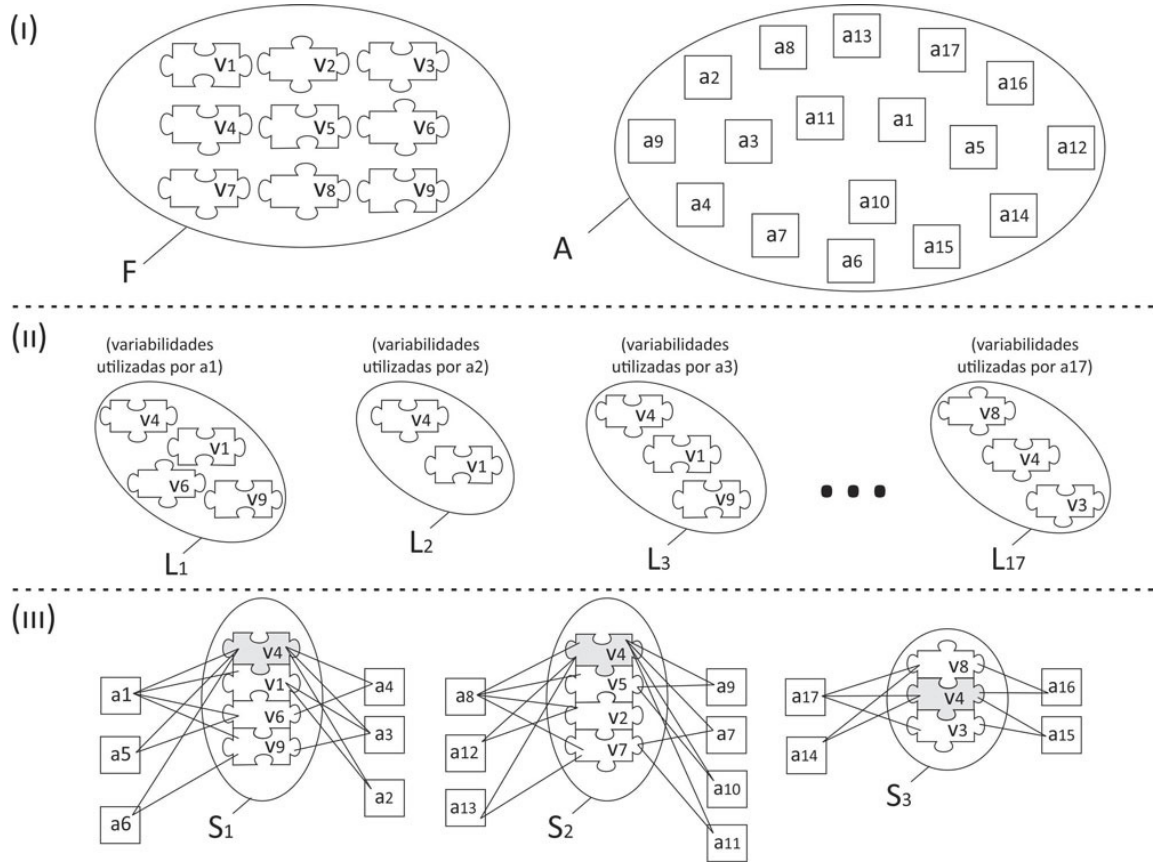


Figura 18: Identificação de subdomínios a partir de um conjunto de aplicações

A partir desses dois conjuntos hipotéticos e considerando a existência de um mapeamento (M) entre as variabilidades do framework e as aplicações, o qual permite identificar quais variabilidades do framework estão sendo utilizadas pelas aplicações, sugere-se adotar os passos descritos no Quadro 5 que estão em forma de algoritmo e auxiliam a identificação dos subdomínios. O algoritmo está dividido em três passos principais.

```
Input: A é um conjunto de aplicações desenvolvidas com F;
       F é um conjunto das variabilidades de um framework;
       M é um conjunto de relações entre elementos de F e A;
Output: S é um conjunto de subdomínios;
Begin
    for todas aplicações (ai...an) do conjunto "A" do
      - Buscar as variabilidades (vi...vn) utilizadas em cada aplicação;
1     - Criar um subconjunto (Li...n) para cada aplicação e adicionar as
        variabilidades encontradas;
    buscar nos subconjuntos (Li...Ln) a frequência em que determinadas
2       variabilidades (vi...vn) estão sendo utilizadas simultaneamente nas
        aplicações (ai...an)
    If é possível agrupar as variabilidades (vi...vn) em
      subconjuntos (Si...Sn) de modo que as variabilidades pertençam
      a um dos subconjuntos then
3       - O framework tem características de um FMD e com base nos
        subconjuntos (Si...Sn), os subdomínios podem ser abstraídos;
    else
      - O framework não é um FMD;
end
```

Quadro 5: Algoritmo para identificação de Subdomínios de um suposto FMD

No passo (1) do algoritmo a análise é iniciada. Para cada aplicação pertencente ao conjunto “A”, devem-se identificar quais variabilidades de “F” estão sendo utilizadas. Com essa identificação, sugere-se criar um subconjunto $L_{i,...,n}$ para agrupar as variabilidades do framework empregadas em cada aplicação. Por exemplo, na parte (II) da Figura 18, o subconjunto L_1 contém todas as variabilidades utilizadas por a_1 e assim por diante.

Nessa atividade, é preferível que as aplicações que tenham evoluído mais vezes e estejam sendo utilizadas por mais tempo, sejam selecionadas primeiro, pois caso uma determinada aplicação estiver utilizando completamente as variabilidades disponibilizadas em “F”, isso indica que os subdomínios, caso existam no framework, não são disjuntos. Mesmo assim, pode-se continuar a análise, pois ainda existe a possibilidade de encontrar aplicações que utilizam variabilidades que se restringem a somente um dos subdomínios que se deseja identificar.

No passo (2) do algoritmo, deve-se buscar nos subconjuntos $L_i,...,L_n$, a frequência em que as variabilidades são utilizadas simultaneamente nas aplicações. Com isso, parte-se para o passo (3), no qual se deve tentar agrupar essas variabilidades formando os subconjuntos $S_i,...,S_n$, de modo que eles sejam suficientes para apoiar os requisitos de um determinado grupo de aplicações, bem como suas evoluções. Caso isso seja possível, pode-se inferir que o framework considerado possui características típicas de um FMD e que a partir dos subconjuntos $S_i,...,S_n$, pode-se abstrair os subdomínios. Caso contrário, o framework considerado não consiste em um FMD e por fim, encerra-se a análise. Nota-se que nessa análise pode ser que existam aplicações que utilizam variabilidades de mais de um subdomínio. Entretanto, podem também existir aplicações que utilizem as variabilidades de somente um deles e que as possibilidades de evolução dessas aplicações se restrinjam a esse único subdomínio. Assim, torna-se coerente separar esses subdomínios, levando em conta esses tipos de ocorrências.

Vale ressaltar que esse agrupamento de variabilidades requer habilidade do EF em notar, por exemplo, que as aplicações: a_1, a_2, a_3, a_4, a_5 e a_6 poderiam evoluir e agregar novas variabilidades, porém essas novas variabilidades pertencem a S_1. Obviamente, algumas variabilidades pertencerão a todos os subconjuntos, afinal todas as aplicações derivadas de um framework podem utilizar uma parte comum, como é o caso da variabilidade v_4, utilizada por todas as aplicações de A. Todavia, se essa parte comum envolver relativamente a maior parte das variabilidades do framework, isso inviabiliza a separação de subdomínios. Variabilidades como v_4 podem ser consideradas como o *Core* da LPF que se deseja obter, sem pertencer a um subdomínio específico.

Na parte (III) da Figura 18, ilustram-se esses subconjuntos, formados a partir da percepção da frequência de utilização das variabilidades contidas em “F”. Apesar das aplicações sofrerem evoluções, não agregam variabilidades externas a esses subconjuntos. Por exemplo, o subconjunto S_1 contém as variabilidades v_1, v_4, v_6 e v_9. Todas essas variabilidades são utilizadas total ou parcialmente por a_1, a_2, a_3, a_4, a_5 e a_6. Essas aplicações utilizam somente as variabilidades restritas a esse subconjunto. Isso indica a existência de subconjuntos que contribuem com a identificação de subdomínios.

De modo geral, essa análise serve apenas como um indicativo de como identificar os subdomínios de um framework, logo não consiste em uma alternativa

definitiva para esse tipo de identificação, pois não se pode garantir que as aplicações não evoluam e utilizem variabilidades até então, não utilizadas e que pertencem a outros subdomínios. A decisão final é com base no conhecimento do domínio (Oliveira Junior et al., 2010).

Com base nessa análise seguindo os passos sugeridos, é importante registrar o mapeamento entre as variabilidades do framework e os subdomínios identificados, mesmo em nível preliminar. Para isso, recomenda-se elaborar um artefato conforme ilustrado no Quadro 6. Cada linha representa um subdomínio e cada coluna, as variabilidades fornecidas pelo framework. Além disso, também deve ser adicionada uma linha para *Core* para o mapeamento das variabilidades comuns. Cada célula marcada indica que certa variabilidade pertence ao subdomínio correspondente. Por exemplo, admitiu-se que as variabilidades: v_2, v_5 e v_7 contribuem com a implementação do S_2 e a variabilidade v_4 pertence ao *Core*, uma vez que pode ser utilizada em ambos os subdomínios.

Variabilidades / Subdomínios	v1	v2	v3	v4	v5	v6	v7	v8	v9
Core									
S1									
S2									
S3									

Quadro 6: Mapeamento entre variabilidades e Subdomínios

Do mesmo modo, essa análise e a elaboração do mapeamento entre as variabilidades e os subdomínios foram realizadas com o GRENJ. Considerando algumas aplicações hipotéticas e a linguagem de padrões que lhe dá suporte, foi possível identificar três subdomínios: vendas, manutenção e aluguel de recursos (*Trade*, *Maintenance* e *Rental*). Esses recursos podem ser: veículos, CDs, reservas em hotel, livros, equipamentos eletrônicos, etc.

No Quadro 7 ilustra-se uma parcela do mapeamento entre as variabilidades disponibilizadas e os subdomínios do GRENJ. Com a elaboração desse artefato, torna-se evidente a existência de variabilidades que são utilizadas de forma restrita em um determinado subdomínio. Por exemplo, a variabilidade “Reserva de Recurso” está mapeada ao subdomínio *Rental*, pois com base no histórico de reuso do GRENJ e nas aplicações avaliadas, não haviam aplicações ligadas à manutenção que permitiam reservar um determinado recurso/componente para um serviço.

Observa-se que além de utilizar o artifício de preencher as células para mapear os subdomínios com as variabilidades, foi inserida a letra inicial de cada subdomínio, para facilitar a análise.

Variabilidades / Subdomínios	Entrega de Recurso	Fornecedor do recurso	Componente utilizada no serviço	Transferência Eletrônica	Cotação de valor	Reserva de Recurso	Venda Associada	Cliente na Transação	Compra de Recurso	Recurso	Compra / Venda de Recurso	Pagamento a vista em dinheiro	Taxa por atraso	Tarefa de Manutenção	Funcionário responsável pela Transação	Item da Transação
Core		C		C				C		C		C	C		C	C
Trade	T				T		T		T		T					
Rental						R	R						R			
Maintenance			M											M		

Quadro 7: Mapeamento entre as variabilidades e os Subdomínios do GRENJ

Uma forma prática para fazer essa análise é mediante o apoio de ferramentas de mineração/extração (Parreira Júnior et al., 2012; Borges, 2010) para automatizar o processo. Nesse caso, pode ser necessário definir textualmente os principais subdomínios, de forma que a ferramenta possa buscar padrões.

5.2.2 Criar Diagrama de features

A segunda atividade do processo, ilustrada pela atividade 1.2 na Figura 17, consiste em **Criar um diagrama de features** que evidencia os subdomínios identificados. Para isso, deve-se ter como entrada o mapeamento entre os subdomínios e as variabilidades do framework, gerado na atividade 1.1. Com isso, para cada linha do mapeamento entre subdomínios e as variabilidades do framework, tem-se uma feature no diagrama de features. No caso da linha referente ao *Core*, a feature respectiva deve ser obrigatória, uma vez que representa os elementos comuns aos subdomínios. Com relação aos subdomínios, devem ser representados por features alternativas, associadas à feature *Core*.

Em geral, considerando que cada feature representa um subdomínio disjunto tratado pelo framework, os membros (produtos) sempre deverão conter apenas uma

feature e a *Core*. Dessa forma, seguindo a ideia de que cada feature representa um subdomínio disjunto, não deve ser possível montar um membro (framework) com mais de uma feature desse primeiro nível de abstração. Assim, todas as features desse nível devem possuir relacionamentos de exclusão umas com as outras, ou seja, um relacionamento alternativo ("*xor*") - mutuamente exclusivo, conforme anteriormente ilustrado no Capítulo 2.

Após a criação desse diagrama e como um passo posterior de refinamento, deve-se verificar se existem configurações válidas que exigem a composição da feature *Core* com mais de um subdomínio, formando um framework que abrange dois subdomínios, por exemplo. Além disso, é necessário analisar as possíveis restrições entre os subdomínios. Na ocorrência de configurações com mais de um subdomínio ou restrições/regras entre eles, deve-se aplicar as devidas mudanças no diagrama.

Na Figura 19 ilustra-se como o diagrama de features (Kang et al., 1990; Barreiros e Moreira, 2011) de alto nível para o GRENJ foi criado na atividade 1.2. Analisando um conjunto de aplicações hipotéticas e as classes do framework, obteve-se inicialmente um diagrama conforme mostrado na parte (a), em que há uma feature *Core* e somente dois subdomínios, representados pelas features: "*Trade+Rental"* e "*Maintenance"*. Essas features estão associadas por meio de um relacionamento mutuamente exclusivo, pois consideramos improvável a existência de aplicações que usam simultaneamente as variabilidades dos subdomínios *Rental* e *Maintenance*. Isso acontece porque o GRENJ foi criado principalmente para servir pequenas empresas, como locadoras. Assim, considerou-se improvável a existência de locadoras que aluguem vídeos/filmes/CDs e também realizem a manutenção desses recursos, como aplicações para bibliotecas. Com relação aos subdomínios *Trade* e *Rental,* puderam ser considerados uma única feature, visto que aplicações para venda (*Trade*) poderiam evoluir agregando a locação (*Rental*) de seus recursos e vice-versa.

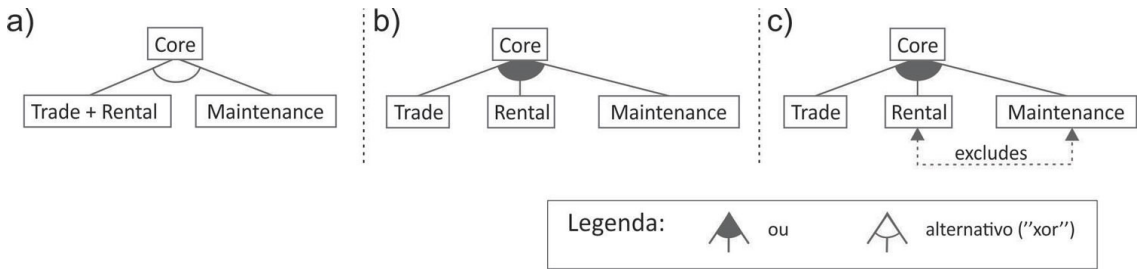


Figura 19: Criação do Diagrama de features que representa os subdomínios do GRENJ

Considerando puramente um conjunto de aplicações e o histórico de utilização do framework admitem-se dois subdomínios. Entretanto, sabendo-se da existência de uma linguagem de padrões e que aplicações que lidam com venda de recursos podem evoluir agregando manutenção, considera-se em nosso estudo de caso, três features conforme mostrado na parte (b) da Figura 19. Para permitir a composição das features *Trade* e *Maintenance*, a associação entre os subdomínios tornou-se "*or*", isso indica que pelo menos uma dentre as features que representam os subdomínios deve ser selecionada. Porém, dessa maneira admite-se que *Rental* e *Maintenance* possam ser compostas em um mesmo framework, quebrando a restrição mencionada anteriormente. Diante disso, o diagrama de features sofreu uma nova modificação, conforme se ilustra na parte (c) da Figura 19, incluindo a restrição (*constraint*) "*excludes*" entre *Rental* e *Maintenance* e permitindo que as features *Trade* e *Maintenance* possam ser compostas.

Assim, uma aplicação para venda e aluguel de filmes pode ser desenvolvida, mas não poderá incluir manutenção. Caso uma aplicação para manutenção de recursos seja criada para uma oficina, a única feature que pode ser adicionada juntamente à feature *Maintenance* é a feature *Trade*. Com relação às features *Trade* e *Rental*, uma aplicação pode ser desenvolvida apenas para venda, como uma aplicação de caixa de supermercado, que não precisa do subdomínio *Rental*. O mesmo acontece na ordem inversa; uma aplicação que trata apenas de aluguel pode não precisar das funcionalidades do subdomínio *Trade,* da maneira como está no diagrama, essas possibilidades são consideradas. Portanto, as configurações válidas admitidas são as seguintes*: "Core + Trade", "Core + Rental", "Core + Maintenance", "Core + Trade + Rental"* e *"Core + Trade + Maintenance".*

Desse modo, aplicações pertencentes ao subdomínio *Trade*, por exemplo, podem utilizar um framework que contém apenas as funcionalidades desse subdomínio, descartando as features remanescentes que estão ligadas aos demais

subdomínios. No processo de reuso do GRENJ, isso não é possível, visto que por mais que as aplicações requeiram somente uma parcela do framework, tais aplicações devem considerá-lo integralmente.

5.2.3 Definir ordem de prioridade para Análise/Modularização

Após a identificação dos subdomínios e a criação de um diagrama de features, opcionalmente parte-se para a atividade 1.3 - **Definir ordem de prioridade para Análise/Modularização**, ilustrada na Figura 17, na qual se define uma ordem de prioridade para analisar e modularizar os subdomínios identificados.

Essa definição pode ser feita mediante necessidades de mercado ou por haver subdomínios com menor dependência com os demais. Assim, exige-se habilidade e conhecimento do domínio pelo desenvolvedor (Oliveira Junior et al.,2010). Considerando necessidades de mercado, por exemplo, pode ser necessário que um subdomínio seja modularizado primeiro para que determinados frameworks possam ser gerados rapidamente. Com relação às dependências, é importante que o EF verifique se existem variabilidades de um dado subdomínio que participam da implementação de um subdomínio diferente. Para isso, sugere-se consultar o mapeamento elaborado na atividade 1.1, em seguida, selecionar uma variabilidade e identificar na documentação em quais unidades essa variabilidade encontra-se implementada. Com isso, pode-se averiguar em nível de código fonte se existem dependências a outra variabilidade mapeada com um subdomínio diferente. Caso tal variabilidade já estiver mapeada com mais de um subdomínio, identificar a dependência torna-se simples, como é o caso da variabilidade “Venda Associada” mostrada no Quadro 7, que pertence ao subdomínio *Rental* e permite que um recurso destinado à locação possa ser utilizado em uma venda.

Com base no mapeamento elaborado na atividade 1.1 e/ou análise do código fonte do framework, recomenda-se elaborar um artefato contendo a ordem em que o subdomínio deve ser analisado em maior nível de detalhamento; sua descrição; e por fim, a informação se há dependência ou não com outro subdomínio, em caso positivo deve-se descrever com qual existe.

No caso do GRENJ, analisando as variabilidades cobertas pelo subdomínio *Maintenance* e as unidades de implementação, não foram encontradas dependências com os demais subdomínios, por isso atribuiu-se a primeira posição,

diferente dos subdomínios *Trade* e *Rental* que possuíam várias variabilidades com dependências entre si.

No Quadro 8 ilustra-se a ordem atribuída para a análise/modularização de cada subdomínio e a existência de dependências entre *Trade* e *Rental*. Vale ressaltar que obviamente todas as features que representam os subdomínios dependem da feature *Core*. No entanto, considerou-se desnecessário adicionar essa informação no artefato. A feature *Core* recebeu a última posição, pois considera-se mesmo que ainda de forma preliminar que essa feature poderia envolver maior esforço.

Ordem	Subdomínio	Dependências com
1º	*Maintenance*	
2º	*Trade*	*Rental*
3º	*Rental*	*Trade*
4º	*Core*	----------

Quadro 8: Prioridade para Análise/Modularização

De modo geral, conforme pode ser observado no Quadro 8, não é necessário detalhar ainda quais são as unidades de implementação que apresentam tais dependências, a ideia se restringe em apenas descobrir em quais subdomínios existem e com isso, se julgar necessário, estabelecer uma ordem que deve-se proceder nas próximas atividades, no que diz respeito a análise e modularização dos subdomínios.

5.2.4 Definir Cenário de Utilização

A próxima atividade do processo é ilustrada na Figura 17 pela atividade 1.4, na qual se deve **Definir o Cenário de Utilização** que a LPF que será obtida no final do processo deve satisfazer. Isso pode estar sujeito ao conhecimento do domínio e a fatores como o perfil de uma organização. Por exemplo, uma organização pode desenvolver aplicações com o apoio do FMD e/ou a organização pode desenvolver frameworks e disponibilizá-los para usuários externos.

Para auxiliar essa definição, descreve-se no Quadro 9 as principais características de cada Cenário de Utilização com relação à LPF e aos seus membros.

	Critério de Comparação	Cenário de Utilização 1	Cenário de Utilização 2
LPF	Granularidade das features	Essencialmente features de granularidade grossa que agregam muitas variabilidades (features em nível de subdomínio)	Essencialmente features de granularidade fina.
	Alvo	Destina-se às organizações de software que desenvolvem e/ou fornecem amplos frameworks a desenvolvedores externos com base em um subdomínio de aplicações e/ou desenvolvem aplicações que geralmente incluem todas ou a maior parte das features disponíveis no framework	Destina-se a organizações de software que fornecem frameworks e/ou desenvolvem aplicações de domínio restrito
Membro da LPF	Relação entre features e variabilidades	Cada **membro** contém poucas features, mas incluem todas as variabilidades de um subdomínio e todas as variabilidades comuns que podem ser usadas em ambos os subdomínios.	Os **membros** podem ser restritos de acordo com os requisitos específicos de cada aplicação. Além disso, pode-se selecionar uma parcela das features que abrangem todos os subdomínios.
	Quantidade de aplicações que podem ser desenvolvidas	Com o apoio de um **membro** pode-se desenvolver muitas aplicações, segundo o subdomínio coberto.	Com o apoio de um **membro** pode-se desenvolver poucas ou somente uma aplicação, como também é possível gerar muitas aplicações, desde que o membro contenha todas as variabilidades do subdomínio.
	Probabilidade de não existir uma determinada feature no membro	Há poucas chances de surgir necessidade por um novo requisito a uma aplicação (desenvolvida com o apoio de um **membro**) que implica em uma variabilidade que não pertence ao membro. Isso ocorrerá somente se tal variabilidade não tiver sido desenvolvida para o subdomínio da LPF	As chances são maiores de surgir demanda por um novo requisito de uma aplicação (desenvolvida com o apoio de um **membro**) que implica em uma feature de granularidade fina inexistente no membro, mas disponibilizada pela LPF. Assim, deve-se buscar essa feature na LPF e associá-la ao membro.
	Quantidade de features não utilizadas	As variabilidades estão associadas às features de granularidade grossa no **membro**. Com isso, muitas variabilidades podem não ser utilizadas por uma aplicação, mas são possíveis futuramente.	A quantidade de features no **membro** pode ser bem restrita e englobam poucas variabilidades. Com isso, a quantidade de features não utilizadas por uma aplicação é mínima ou praticamente nula.

Quadro 9: Características dos Cenários de Utilização de uma LPF

Conforme discutido anteriormente, identificamos dois Cenários de Utilização sob o ponto de vista do uso de uma LPF, uma vez que o uso do framework (membro) é o mesmo. No Cenário de Utilização 1, a LPF possui poucas features mas agregam uma grande quantidade de variabilidades. Essas features são de granularidade grossa e ao desenvolver uma aplicação instanciando um membro resultante da composição dessas features, é provável que a maioria das variabilidades disponíveis não sejam utilizadas, mas são possíveis futuramente. No Cenário de Utilização 2, a LPF possui features de granularidade fina. Com essa LPF, podem-se compor membros mais direcionados e restritos, segundo os requisitos de uma aplicação, evitando features desnecessárias. Porém, essa LPF pode despender mais tempo para selecionar as features para um membro (framework).

Em nosso estudo de caso, optou-se pelo Cenário de Utilização 2 para averiguar todo o processo de modularização. Além disso, como esse cenário inclui o Cenário de Utilização 1, ambos estariam sendo avaliados.

5.2.5 Selecionar feature para Análise

A quinta atividade do processo, ilustrada pela atividade 1.5 na Figura 17, consiste em **Selecionar feature para Análise**. Para isso, deve-se ter como entrada o artefato que descreve a ordem de prioridade para Análise/Modularização dos subdomínios. A partir dessa atividade são iniciadas iterações com as features selecionadas até atingir a atividade 2.1 - **Definir Técnica para modularização de features**. Em cada iteração uma feature é selecionada e analisada completamente, identificando as unidades que colaboram com sua implementação. Para concluir essas iterações, todos os subdomínios devem ser analisados, incluindo a parte comum do framework que apesar de não ser analisado como um subdomínio pode ser considerado como uma feature que possui um nível de abstração maior, assim como os subdomínios.

5.2.6 Identificar unidades de implementação

Depois de selecionar a feature na atividade 1.5, parte-se para a atividade 1.6 - **Identificar unidades de implementação,** ilustrada na Figura 17. Nessa atividade deve-se analisar as variabilidades fornecidas pelo framework com o objetivo de identificar as unidades que participam da implementação da feature selecionada. Para isso, deve-se ter como entrada o artefato de mapeamento entre as variabilidades e subdomínios identificados que foi elaborado na atividade 1.1. Sabendo-se quais variabilidades existem, pode-se procurar quais unidades colaboram com sua implementação.

Na Figura 20 ilustram-se trechos de código de duas classes do framework GRENJ. Esses fragmentos servem para exemplificar a identificação de unidades que colaboram com a implementação de uma feature selecionada na atividade (1.5). Nesses trechos de código, retratam-se como tais classes participam da implementação das features *Core*, *Trade* e *Rental*. Muitas unidades de

implementação do framework possuem trechos de código entrelaçados, dificultando essa identificação.

Na parte esquerda da Figura com a descrição “Classe ResourceRental”, mostra-se um dos construtores dessa classe. Nota-se que no construtor da classe *ResourceRental* há um trecho condicional “`if`” para “`this.hasAssociatedSale()`” que apesar de se tratar de um método abstrato declarado na classe *ResourceRental*, não está diretamente ligado ao subdomínio *Rental*. Trata-se de um método que deve ser concretizado, adicionando um “`return true;`”, caso uma determinada aplicação de aluguel (*Rental*) requerer Venda associada aos recursos. Desse modo, pode-se admitir que essa unidade de implementação colabore com a implementação das features *Trade* e *Rental*.

Na parte direita da Figura 20 com a descrição “Classe SingleResource”, mostra-se alguns métodos dessa classe. Com base na observação de algumas aplicações desenvolvidas com o apoio do GRENJ e na navegação pelo código do framework, notou-se a importância dessa classe para todas as aplicações, por isso a notação lateral “*Core*”, indicando que essa classe deverá ser obrigatória para todos os membros da LPF. O primeiro método “`calculateAvailableQttyToRent`”, serve para retornar a quantidade de recursos disponíveis para locação, logo entende-se que esse método colabora com a implementação do subdomínio *Rental*. O segundo método “`calculateAvailableQttyToSell`” tem a mesma função que o método “`calculateAvailableQttyToRent`”, porém é utilizado para aplicações que envolvem venda. Por fim, o método “`checkAvailabilityToRentQtty`” é utilizado para verificar a disponibilidade de um determinado recurso para locação, retornando um valor *booleano*.

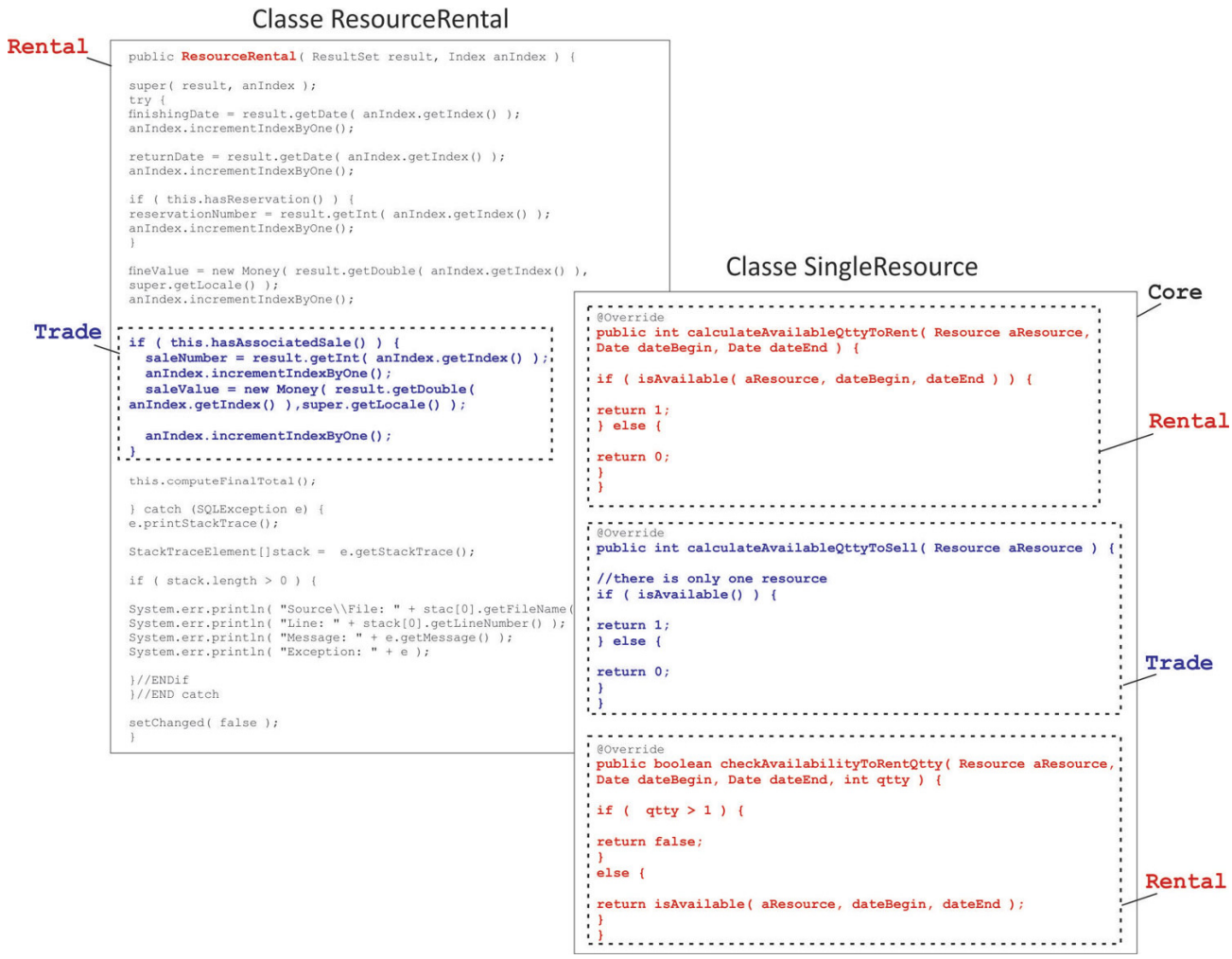


Figura 20: Identificação de unidades de implementação das features

Com a identificação das unidades que colaboram com a implementação da feature, parte-se para a atividade 1.6.1 - **Elaborar/Refinar Mapeamento entre unidades de implementação e features**, ilustrada na Figura 17, na qual se sugere desenvolver um artefato para associar as unidades de implementação com sua(s) feature(s) correspondente(s). Para que tais unidades sejam associadas, basta haver algum trecho de código que remeta à implementação da feature. Esses fragmentos de código podem ser métodos, atributos, chamadas a métodos externos, etc.

Para cada feature selecionada na iteração de análise, deve ser criada uma linha nesse mapeamento e para cada unidade de implementação analisada, uma coluna deve ser adicionada. Para indicar a relação entre unidades e a feature marca-se a célula nesse mapeamento. No Quadro 10 mostra-se uma parcela desse *Mapeamento entre features e unidades de implementação* utilizado na modularização do GRENJ. As linhas contêm as features da LPF que são referentes aos subdomínios e ao *Core*. Nas colunas, a descrição das unidades de

implementação. Cada célula preenchida indica que a unidade colabora com a implementação da feature. Por exemplo, a classe *AbstractCalculator* pode ser utilizada por aplicações de ambos os subdomínios, por isso foi mapeada como uma unidade que participa da implementação da feature *Core*.

Classes / Features	AbstractCalculator	BasicDelivery	BasicMaintenance	BasicNegotiation	BasicPurchase	BasicSale	BusinessResourceQuotation	BusinessResourceTransaction	Cash	CashOnDelivery	ConnectionManager	CreditCard	creditCardValidador	DestinationParty	EletronicTransfer	ExactNumberCalculator
Core																
Trade																
Rental																
Maintenance																

Quadro 10: Mapeamento entre features da LPF e unidades de implementação do GRENJ

5.2.7 Decompor feature em nível mais granular

Conforme ilustrado na Figura 17, a partir da atividade 1.5 pode-se seguir para a atividade 1.7 - **Decompor feature em nível mais granular**, caso a motivação seja obter uma LPF cujas features possuem granularidade fina. Essa atividade é caracterizada pela decomposição da feature selecionada na atividade 1.5, que pode ser referente a um dos subdomínios ou a *Core*. Para isso, deve-se ter como artefato de entrada, o mapeamento entre as variabilidades e os subdomínios identificados, gerado na atividade 1.1.

A partir desse mapeamento, as variabilidades devem ser analisadas para decidir quais variabilidades devem ser consideradas como novas features. Para isso, o conhecimento do domínio é fundamental. Com base nessa decisão, parte-se para a atividade 1.7.1 - **Refinar diagrama de features**, na qual as novas features são adicionadas ao diagrama elaborado na atividade 1.2.

Na atividade 1.7.1, a feature selecionada na atividade 1.5 deve ser dissociada em features de granularidade fina, caso possível. Para isso, devem-se analisar em nível funcional as variabilidades do framework, investigando quais podem se tornar

features opcionais, obrigatórias, associadas por meio de um relacionamento "*or*" ou alternativas (*xor*).

Segundo Kang et al. (2002), uma feature pode ser considerada opcional quando puder ou não estar presente nos produtos, obrigatória quando necessária para todos os produtos e alternativa ("*xor*"), quando dado um subconjunto de features, exatamente uma puder ser selecionada para o produto. Com relação às features associadas com um relacionamento "*or*", são opcionais e estão agrupadas de acordo com uma determinada categoria ou similaridade e não são mutuamente exclusivas.

Uma feature opcional pode ser removida sem impactar na semântica do núcleo de um determinado software. No caso do framework, para qualquer fluxo de execução do mesmo, a remoção desse tipo feature não interfere nas condições esperadas pelos trechos de código contidos no framework, de modo que continue em operação. No caso de uma feature obrigatória, sua remoção impede que o framework possa ser utilizado sem a ocorrência de erros e falhas. Assim, admite-se que a feature *Core* pode encapsular somente essas funcionalidades obrigatórias. Com relação à definição das features que podem ser associadas por meio de um relacionamento "*or*" ou alternativo (*xor*) - mutuamente exclusivo, devem-se considerar quais features podem ser agrupadas em uma categoria. A partir disso, deve-se averiguar se uma ou todas as features pertencentes a essa categoria podem ser selecionadas (relacionamento "or") ou a seleção de uma, deve impedir a seleção das demais (alternativo "xor"), considerando a geração de membros da LPF.

As variabilidades mapeadas para cada subdomínio na atividade 1.1 devem ser analisadas e com base no conhecimento do domínio do framework, decidir quais podem ser dissociadas como features. Com a definição dessas features, deve-se refinar o diagrama obtido na atividade 1.2. No caso do GRENJ, partiu-se das definições dos subdomínios do framework, ilustrado na parte (c) da Figura 19 e do mapeamento entre as variabilidades e os subdomínios, conforme ilustrado no Quadro 7.

Com base na análise das aplicações desenvolvidas com o apoio do framework, admitiu-se que se uma determinada feature na maioria dos casos não é utilizada, o EF pode decidir tratá-la como uma feature opcional, caso contrário, pode permanecer obrigatória, inclusa na feature *Core*. Na Figura 21 ilustra-se o diagrama de features com as features dissociadas de cada subdomínio. Desse modo, as

novas features podem ser consideradas ou não nos membros gerados, permitindo uma quantidade maior de membros direcionados aos requisitos específicos das aplicações.

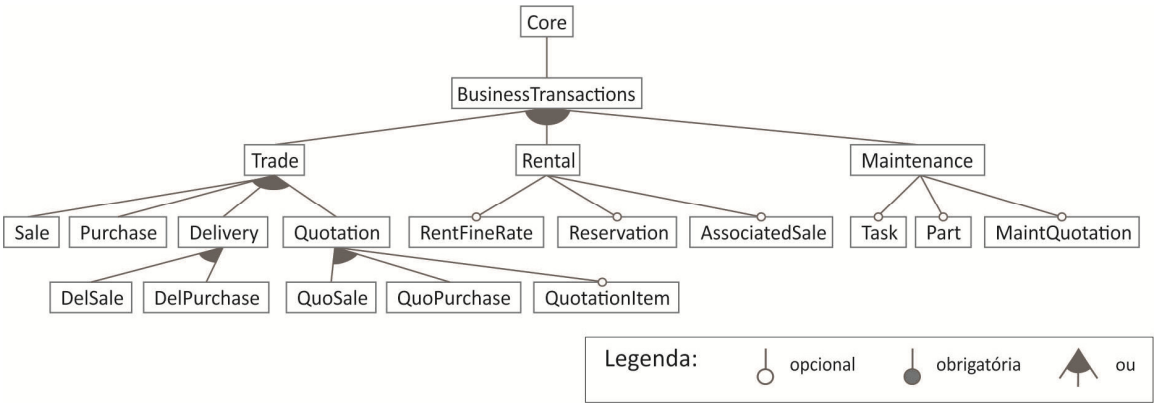


Figura 21: Features externalizadas dos subdomínios do GRENJ

Nesse refinamento do diagrama, deve-se levar em conta as dependências entre os subdomínios. No caso do GRENJ, percebeu-se inicialmente a existência de algumas dependências entre as features *Trade* e *Rental*. Conforme comentado anteriormente, a funcionalidade denominada "*Associated Sale*" está ligada ao subdomínio *Trade*, porém encontra-se implementada na classe *ResourceRental* que é uma das unidades que colaboram com a implementação da feature *Rental*, conforme ilustrado na Figura 20. Na figura 21, pode-se visualizar que "*Associated Sale*" tornou-se uma feature opcional e sua dependência ao subdomínio *Trade* pôde-se restringir a feature *Sale* que pertence a *Trade*, ou seja, pode-se criar uma regra do tipo *AssociatedSale* "implies" Sale, para que ao selecionar a feature *AssociatedSale* a feature *Sale* seja também selecionada.

Na Figura 22 ilustra-se um diagrama que contém as features que podem ser utilizadas por aplicações de todos os subdomínios cobertos pela LPF. Na definição dessas features que são comuns aos subdomínios, perceberam-se certas particularidades no diagrama de features para representar uma LPF. Conforme ilustrado na Figura 22, ligado a feature "*ResourceQuantification*" há quatro features com uma associação do tipo "*or*", com isso, o membro pode conter todos os tipos de quantificações para o recurso. Todavia, ao instanciar o framework GRENJ, define-se apenas um tipo de quantificação para o recurso, pois não é possível utilizar várias quantificações para o mesmo, salvo casos em que na mesma aplicação existam vários tipos de recursos e necessariamente cada recurso tenha um tipo diferenciado

de quantificação. Mas considerando o histórico de utilização do GRENJ não há registros dessa ocorrência.

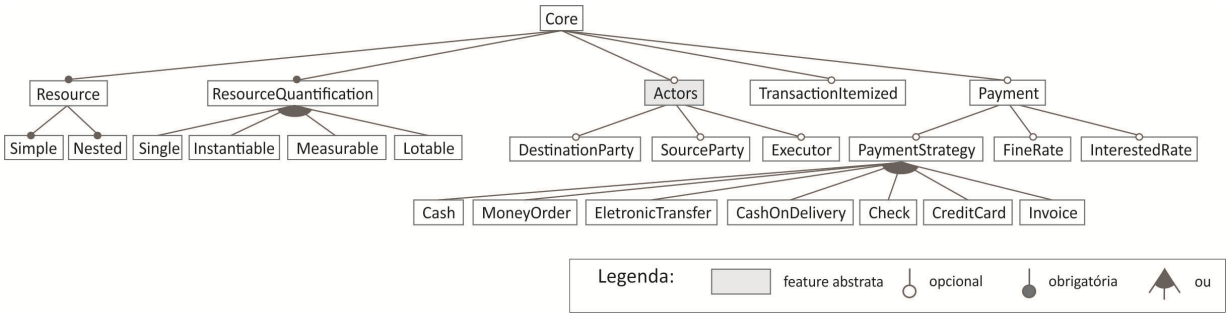


Figura 22: Features externalizadas da feature *Core*

Sob uma visão das LPS, se uma aplicação desenvolvida com o apoio do GRENJ, utiliza apenas um tipo de quantificação, o relacionamento entre as features não deveria ser do tipo "*or*", mas um relacionamento alternativo (*xor*) para que somente uma das quantificações pudesse ser selecionada. No entanto, como os membros de uma LPF não são aplicações concretas, mas frameworks, pode-se permitir a geração de membros com todas essas possibilidades de quantificação. Dessa maneira, o framework gerado a partir da linha pode conter de um a todos os tipos de quantificações.

De modo geral, sugere-se analisar esse tipo de ocorrência com relação às features alternativas, pois com base no conhecimento de utilização do FMD, pode ser coerente um relacionamento do tipo "*or*" entre um determinado subconjunto de features, ao invés de um relacionamento mutuamente exclusivo. Dessa forma, torna-se possível selecionar de uma até todas as features desse subconjunto. Afinal, o membro da LPF consiste em um framework e o EF pode julgar necessária a presença desse subconjunto no framework, mesmo que uma dada aplicação utilize inicialmente somente uma dessas features.

Quanto à feature "*Resource*" e as features "*Simple*" e "*Nested*" que descrevem as categorias possíveis para o recurso, são obrigatórias e por isso poderiam continuar "inclusas" na feature *Core*, mas assumindo que o EF deseja visualizá-las e manter o mapeamento entre as features e as classes que a implementam, optou-se por mantê-las no diagrama.

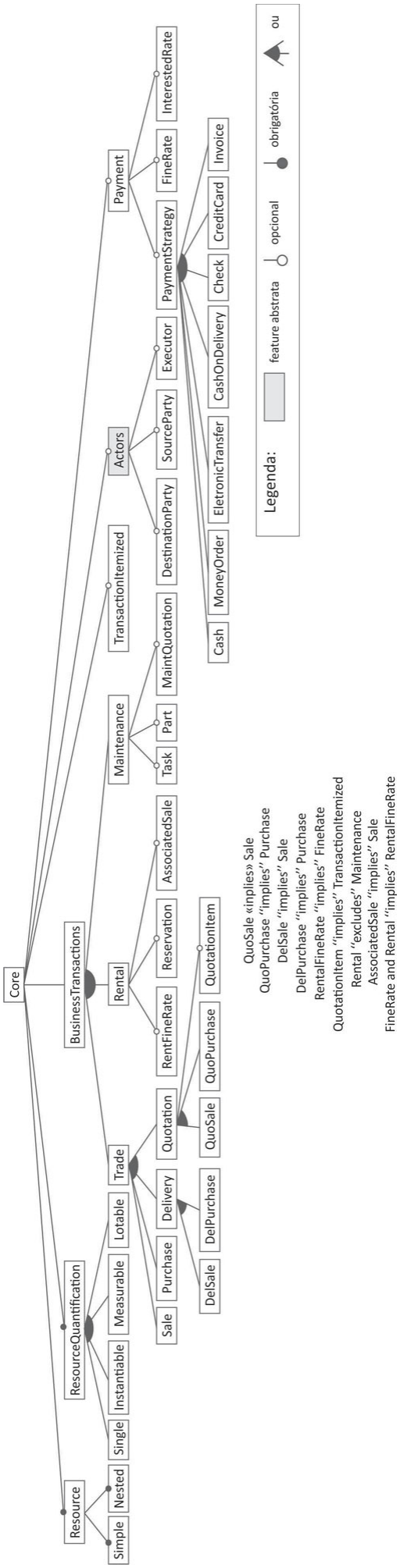


Figura 23: Diagrama de features do GREN segundo o Cenário de Utilização 2

Na Figura 23, ilustra-se o diagrama completo de features da LPF segundo o Cenário de Utilização 2. Nesse diagrama, encontra-se a sobreposição dos diagramas das Figuras 21 e 22, contendo as restrições entre as features.

Com o diagrama de features refinado na atividade 1.7.1, parte-se para a atividade 1.6, na qual se devem identificar as unidades de que colaboram com a implementação da features selecionadas e em seguida na atividade 1.6.1 - **Elaborar/Refinar Mapeamento entre as unidades de implementação e as features**, conforme discutido anteriormente.

A diferença nesse mapeamento, considerando a intenção de obter uma LPF segundo o Cenário de Utilização 2 e ter passado pelas atividades 1.7 e 1.7.1 é que a quantidade de features é maior, uma vez que as features não são apenas as referentes aos subdomínios e ao *Core*, como também as features que foram externalizadas. No Quadro 11 descreve-se parte desse mapeamento, incluindo as features externalizadas. Nota-se que existem células marcadas com um “tom” de cinza mais escuro e aquelas com um “tom” fraco. No primeiro caso, esse artifício foi empregado para indicar as unidades que em sua totalidade ou a maior parte do código remete a feature. No segundo caso, a unidade contém poucos fragmentos de código ligados a feature, como chamadas a métodos externos, atributos, etc. Dessa maneira, pode-se visualmente indicar ainda de forma preliminar, as unidades que mais contribuem com a implementação de uma determina feature.

Classes / Features	AbstractCalculator	BasicDelivery	BasicMaintenance	BasicNegotiation	BasicPurchase	BasicSale	BusinessResourceQuotation	BusinessResourceTransaction	Cash	CashOnDelivery	ConnectionManager	CreditCard	creditCardValidador	DestinationParty	EletronicTransfer	ExactNumberCalculator
Core	■										■		■			■
ResourceQuantification																
Single								■								
Instantiable								■								
Measurable						■		■								
Lotable						■		■								
BusinessTransactions																
Trade		■		■												
Sale						■										
Purchase					■											
Delivery		■			■	■										
DelSale																
DelPurchase																
Quotation							■									
QuoSale																
QuoPurchase																
QuotationItem							■									
Rental																
RentalFineRate																
Reservation																

Quadro 11: Mapeamento entre unidades de implementação e features (em nível mais granular) da LPF

5.3 Fase de Modularização de features

A fase de Modularização concentra o maior esforço do processo, uma vez que se baseando no artefato de mapeamento entre unidades de implementação e as features e diagrama de features, elaborados na fase de análise, deve-se em nível de código implementar as features de modo que, posteriormente seja possível combiná-las gerando frameworks válidos a partir da LPF final.

5.3.1 Definir técnica de modularização de features

Apesar de o processo ser independente de uma técnica/plataforma específica, deve-se **Definir uma técnica para modularização das features**, conforme descrito na atividade 2.1 ilustrado na Figura 17. Por meio dessa técnica, deve-se introduzir e refinar comportamentos das unidades de implementação conforme a composição das features.

Nesse sentido, averígua-se duas tecnologias: POA com a linguagem AspectJ e MDD com templates Acceleo. A POA foi escolhida por ser uma técnica bem conhecida para a modularização de interesses e AspectJ por ser uma linguagem disseminada nessa modalidade. Assim, como o processo apresentado envolve a modularização de interesses (subdomínios), consideramos a AspectJ como bom candidato. Acceleo foi selecionado pois considera os recentes avanços no campo M2T. Além disso, essa ferramenta fornece facilidade para iniciantes quanto à compreensão de seus princípios básicos, em termos de geração de código a partir de modelos, um dos pontos envolvidos no MDD. Essa ferramenta permite associar trechos de código com suas respectivas features. Assim, também consideramos essa ferramenta para apoiar a modularização.

Ambas as tecnologias foram utilizadas na obtenção das LPFs conforme os dois cenários de utilização mencionados anteriormente. Com orientação a aspectos foi possível introduzir comportamentos e propriedades às classes, de maneira estática e dinâmica. Com templates Acceleo, os trechos de códigos do framework foram associados com suas features correspondentes. Nas subseções seguintes são apresentados os detalhes de como modularizar features com ambas as tecnologias.

5.3.2 Selecionar feature para Modularização

Na atividade 2.2 ilustrada na Figura 17, deve-se **Selecionar a feature para a iteração atual de modularização**. Nessa atividade, as features selecionáveis podem ser somente as referentes aos subdomínios e ao *Core* – se a motivação consistir em obter uma LPF que atenda a ideia do Cenário de Utilização 1, como também pode incluir todas features que foram dissociadas – caso a motivação seja

obter uma LPF mais granular de acordo com a ideia do Cenário de Utilização 2, conforme discutido anteriormente.

Para essa seleção, pode-se considerar novamente o artefato referente à Ordem de Prioridade para Análise/Modularização de Subdomínios, elaborado na atividade 1.3, para que o subdomínio de maior prioridade seja modularizado primeiro e assim por diante.

Após a seleção da feature na atividade 2.2, parte-se para a atividade 2.3 - **Modularizar fragmentos de código e implementar a feature**, conforme ilustrado na Figura 17. Para essa modularização, deve-se consultar o mapeamento entre features e unidades de implementação, criado na atividade 1.6.1. Dessa forma, podem-se verificar quais unidades são afetadas e quais colaboram integralmente com a implementação da feature em questão.

Com relação à utilização de POA, cada feature pode ser modularizada como um ou mais aspectos (Zhang e Jacobsen, 2004). A partir disso, ao compor as features gerando membros da LPF, o desenvolvedor poderá selecionar os aspectos que deseja incluir na compilação. Com a criação dessas novas unidades de implementação, sugere-se elaborar um mapeamento adicional entre as features e os aspectos (*.aj) desenvolvidos e em seguida, refinar o artefato responsável pelo mapeamento entre as features e as unidades de implementação. Segundo Kästner et al. (2008), essa técnica é mais adequada para tratar features de granularidade grossa, visto que os aspectos são projetados para refinar pontos pré-definidos da execução de um determinado software. Isso se aplica a LPF cujas features possuem um nível maior de abstração e agregam um grande conjunto de variabilidades, como é o caso de uma LPF segundo a ideia do Cenário de Utilização 1.

No caso das features de granularidade fina, que envolvem código no interior de métodos, pode ser exigido um esforço considerável (Kästner et al., 2007). Por exemplo, utilizar amplamente *advices* do tipo *around* para entrecortar uma grande quantidade de métodos adicionando refinamentos, entrecortar muitos construtores para adicionar atributos no objeto e depois retorná-los utilizando o mecanismo *proceed()*, etc. Além disso, para levar em conta as possíveis combinações entre as feature, muitas vezes, deve-se criar uma quantidade exponencial de aspectos que possivelmente conterá replicação de código. Desse modo, muitos desses aspectos podem afetar o mesmo ponto de junção, exigindo que o EF estabeleça uma ordem

de precedência. Esse caso se aplica a LPF conforme a ideia do Cenário de Utilização 2.

Para a modularização de features com o Acceleo, deve-se migrar completamente o código do framework para templates. Desse modo, cada template pode ser utilizado para a geração de uma classe ou outro tipo de unidade de implementação. Para a definição das features em termos de código fonte, devem-se criar regras de composição nos templates por meio de *tags*, indicando quais trechos do código colaboram com a implementação de uma determinada feature. Obviamente, para gerar o código referente às features selecionadas, é necessário partir de um modelo abstrato. Esse processo de associação de trechos de código à features é similar a utilização da ferramenta CIDE (Kästner et al., 2007), porém ao invés de *tags*, permite associar cores de fundo à linhas de código que implementam uma feature.

No caso do GRENJ, foi criado um modelo de features utilizando a ferramenta de modelagem de FeatureIDE. Nesse modelo, além de incluir as features com suas propriedades e associações, foram criadas as regras/restrições que são imprescindíveis para a criação de produtos válidos. A partir desse modelo, foi possível selecionar as features e criar configurações. Porém, isso ainda não foi suficiente para a geração de código com o Acceleo. Com isso, foi desenvolvido um metamodelo com EMF que descreve a estrutura das features de forma aceitável para o Acceleo e um *plug-in* para a IDE Eclipse que transforma configurações válidas de modelos do FeatureIDE em uma instância desse metamodelo. Esse *plug-in* e o metamodelo podem ser utilizados na modularização de outros frameworks. Com base na configuração das features do modelo, uma instância (*.*xmi*) do metamodelo pôde ser criada pelo *plug-in* desenvolvido. Com essa instância, o Acceleo torna-se capaz de gerar as classes do membro da LPF, conforme as features selecionadas.

O esforço empregado para modularizar as features referentes aos subdomínios e ao *Core* do GRENJ, com as duas tecnologias foi relativamente equivalente. Com AspectJ, foram utilizadas somente declarações *inter tipo* para introduzir os métodos nas classes adequadas e tratar as possíveis dependências entre os subdomínios. Nos templates Acceleo, foram utilizados somente "`ifs`" do tipo "`[if(feature.name.equalsIgnoreCase('Trade'))]`" rotulando os métodos e atributos que pertenciam aos subdomínios. Desse modo, foram obtidas duas LPFs

segundo o Cenário de Utilização 1 a partir do GRENJ, uma com AspectJ e outra com Acceleo, que permitiam gerar membros segundo as cinco combinações válidas, descritas anteriormente na subseção 5.2.2.

Para obter uma LPF segundo o Cenário de Utilização 2, as features por serem mais granulares, podem afetar uma quantidade maior de unidades de implementação. Isso abrange, por exemplo, o tratamento de construtores e diversas classes com trechos de código que envolvem múltiplas features. Para modularizar essas features por meio de orientação a aspectos foram necessários vários aspectos com vários advices do tipo *around* e declarações *inter tipo*, resultando em uma grande quantidade de aspectos para considerar as combinações válidas entre as features. Além de ser um processo mais lento, foi necessário atentar para a precedência dos aspectos que entrecortam os mesmos pontos de junção das classes, impedindo que houvesse interferências. A utilização de templates foi mais eficiente, pois da mesma forma que no Cenário de Utilização 1, todos os métodos, atributos e refinamentos de métodos foram rotulados facilmente com “ifs”, associando cada fragmento de código à sua feature correspondente. Vale ressaltar que essa marcação manual dos trechos pertencentes a cada feature, em frameworks de grande porte, mostra-se tediosa e sujeita a erros, exigindo atenção e amplo conhecimento. Um ponto positivo da utilização de templates para obtenção de uma LPF é que essa associação pode ser feita em FMDs desenvolvidos em diversas linguagens de programação, possibilitando gerar código em qualquer extensão. De modo geral, foram obtidas outras duas LPFs, permitindo gerar membros com um conjunto de features mais restrito e direcionado aos requisitos das aplicações.

5.3.3 Compor features e gerar membros da LPF

Com a feature modularizada na atividade 2.3, parte-se para a atividade 2.4 - **Compor features e gerar membros da LPF**, conforme descrito na Figura 17. Nessa atividade, deve-se compor a feature com as previamente implementadas, até que todas as features tenham sido consideradas. A maneira utilizada para composição dessas features para gerar membros da LPF está sujeita à tecnologia empregada.

No caso do GRENJ, ao compor as features implementadas com AspectJ e as mesmas com Acceleo, notou-se diferenças em termos de garantir a criação de membros válidos.

A composição das features modularizadas com AspectJ tem alta tendência a erros, uma vez que o EF pode selecionar aspectos indevidamente ou deixar de selecionar os aspectos importantes para o membro que se deseja. Por exemplo, as propriedades de cada feature (obrigatória, opcional, relacionamento "*or*" ou relacionamento mutuamente exclusivo "*xor*") e as regras/restrições para criar membros válidos, ficam restritas em um diagrama e serve apenas para guiar a composição dessas features, servindo como fonte de auxílio (documentação). Desse modo, nada impede que o EF selecione acidentalmente uma feature (aspecto), bem como desrespeite regras criadas, como por exemplo, ao escolher a feature "A", deve-se selecionar as features "G" e "H". Como consequência disso, pode haver interferências entre os aspectos e erros em relação às dependências.

Com a utilização de templates Acceleo, a composição inicia de um modelo que possui as propriedades das features e as restrições que garantem a criação de membros válidos e coerentes. Esse modelo não serve apenas como fonte de auxílio, como também um artefato utilizado para geração de código. Dessa maneira, as unidades de implementação são geradas e/ou refinadas baseando-se nas configurações válidas possíveis, que incluem as propriedades, associações, regras do tipo "*requires/implies*" e "*excludes"*, etc. Isso evita os problemas descritos anteriormente quanto à composição de features utilizando aspectos.

Nas Figuras 24 e 25, ilustra-se passo a passo como criar uma configuração a partir de um modelo de features, realizar a composição das features e por fim, obter o código correspondente, por meio da ferramenta de modelagem FeatureIDE, o *plug-in* desenvolvido e templates Acceleo.

Na parte 1 da Figura 24 mostra-se o diagrama de features da LPF obtido segundo o Cenário de Utilização 2, trata-se do mesmo diagrama mostrado na Figura 23. Esse modelo foi desenvolvido com apoio da ferramenta FeatureIDE. Observa-se que as features estão associadas a partir de uma feature no topo que representa o *core* da LPF. Na parte inferior descrevem-se as regras necessárias para a criação de configurações válidas. Na parte 2 ilustra-se uma tela do ambiente Eclipse, na qual se deve escolher a opção *FeatureIDE >> Cofiguration File*. Isso serve para criar um arquivo de configuração a partir do modelo de features. Com a escolha dessa opção, é necessário definir um nome para esse arquivo. No caso ilustrado, o nome escolhido foi "`conf`" e como a extensão desse arquivo é "`.config`", resultou em "`conf.config`", conforme ilustrado no topo da parte 3 da Figura.

Na parte 3 é possível visualizar a estrutura das features de uma maneira diferente. Cada feature selecionável (não obrigatória) possui um "*check box*" que permite a escolha. Como se pode observar, dentre as features selecionadas estão: *Instantiable, Trade, Sale, SourceParty, DestinationParty,* etc.

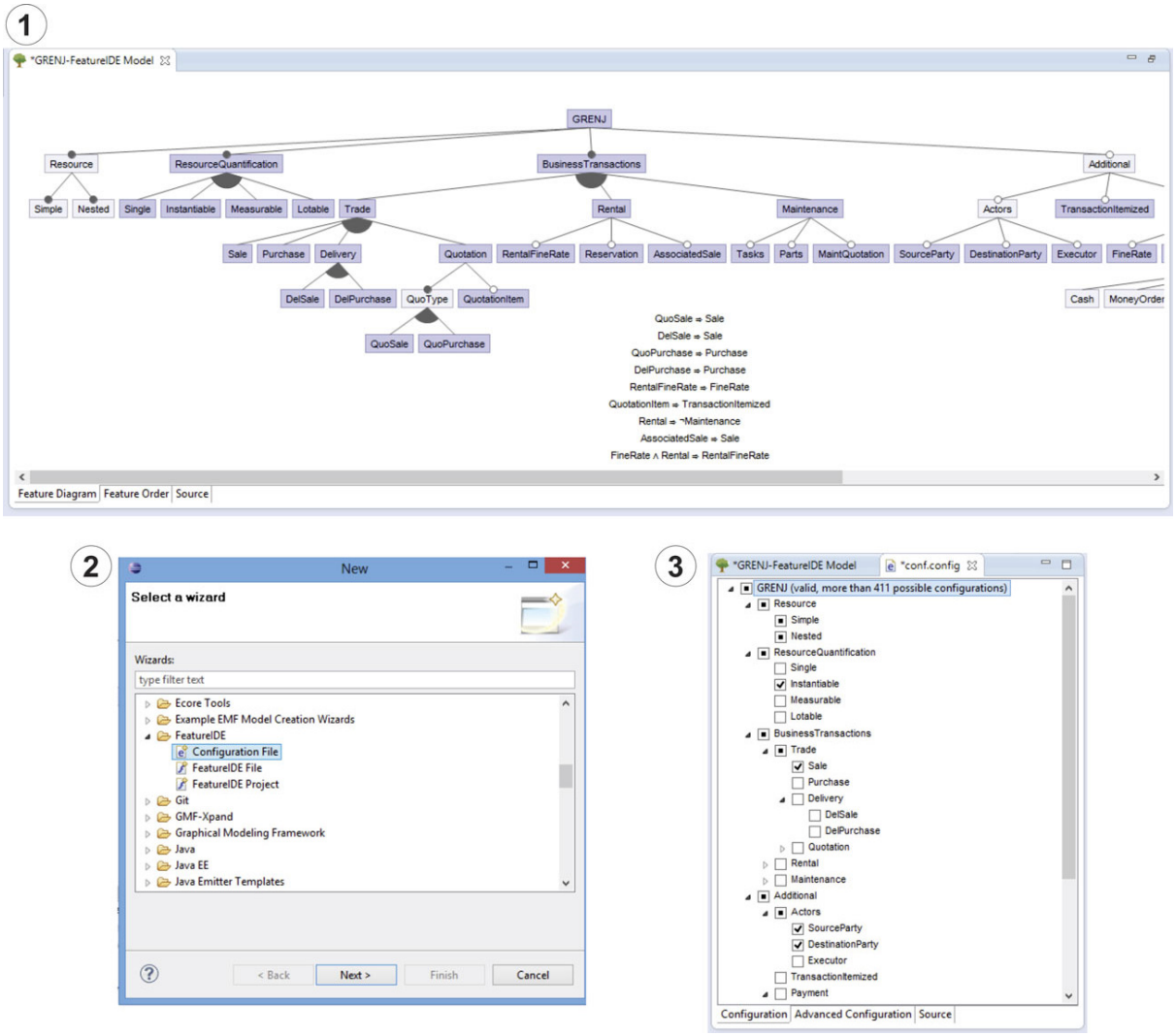


Figura 24: Criação de uma configuração a partir do modelo de features com FeatureIDE

Na Figura 25, ilustra-se a continuidade da composição de features iniciada com o apoio da Figura 24. A partir da configuração, do modelo de features e dos templates criados previamente, o gerador do Acceleo é capaz de fornecer unidades de implementação refinadas/modificadas com somente as features escolhidas para o membro da LPF. Para isso, um *plug-in* foi desenvolvido com o intuito de analisar o modelo de features e o arquivo de configuração e com isso, criar uma instância (**.xmi*) de um metamodelo para servir como entrada para o Acceleo, conforme discuto anteriormente. A partir dessa instância, o Acceleo é capaz de gerar código

com base nos templates desenvolvidos previamente, que por sua vez possuem as devidas associações com as features.

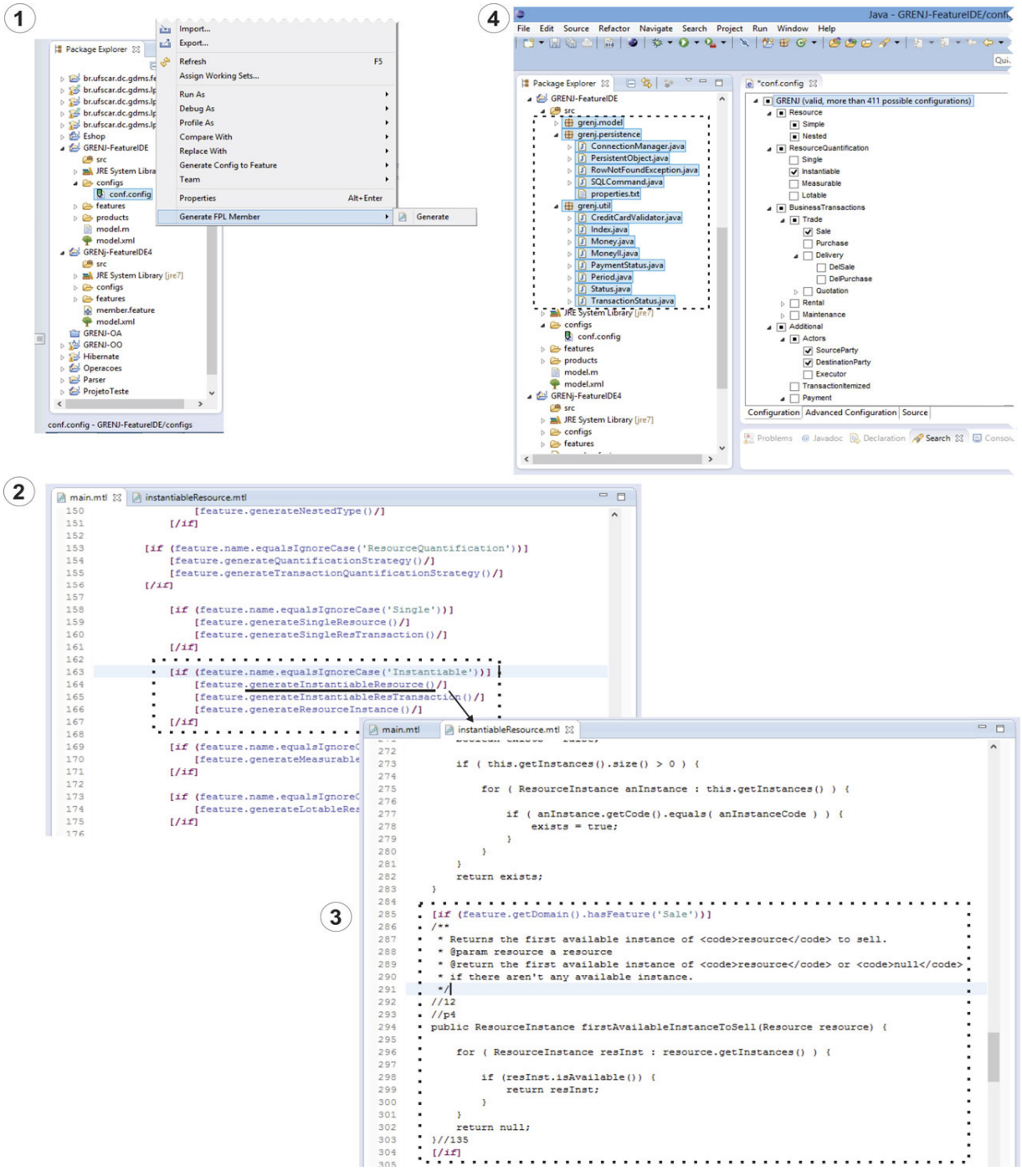


Figura 25: Geração de código de um membro da LPF com Acceleo a partir do modelo de features e uma configuração válida

Na parte 1 da Figura 25 mostra-se o *plug-in* sendo executado pelo usuário. Com um clique sobre o arquivo de configuração, apresenta-se uma opção para gerar o membro da LPF correspondente as features escolhidas. Com essa execução, as

unidades de implementação são geradas, conforme ilustrado na parte 4 da Figura que está delimitada por um retângulo com linhas tracejadas.

Como já mencionado, para que o Acceleo gere essas unidades, é necessário a implementação dos templates. Na parte 2 da Figura 25, mostra-se um trecho de um template definido como principal, denominado "`main.mtl`", visto que a partir dele são realizadas chamadas a outros templates. Observa-se que há um trecho delimitado por um retângulo com linhas tracejadas. Nesse trecho delimitado pela *tag* "`[if(feature.nome.equalsIgnoreCase('Instantiable'))]`" e "`[/if]`". Na parte interna desse trecho, existem algumas instruções. Dentre elas, há uma que está sublinhada "`[feature.generateInstantiableResource()/]`". Essa instrução serve para gerar a classe *InstantiableResource.* O template responsável por essa classe denominado "`instantiableResource.mtl`", é mostrado parcialmente na parte 3 da Figura 25. Nota-se que há outro trecho de código delimitado por um retângulo com linhas tracejadas e na parte interna desse retângulo há uma instrução "`[if (feature.getDomain().hasFeature('Sale'))]`" finalizada com a tag "`[/if]`". A ideia é que qualquer configuração que contiver as features *Instantiable* e *Sale*, esse bloco de código pertencerá à Classe *InstantiableResource*.

De modo geral, ao modularizar todas as features, obtém-se uma LPF conforme o Cenário de Utilização definido no início do processo. Nota-se na Figura 17 há uma seta partindo da atividade 2.4 que alcança a atividade 2.2, isso serve para ilustrar que enquanto houver features a serem modularizadas, deve-se continuar a modularização.

No caso do GRENJ, após todas as iterações de modulação possíveis, obteve-se uma LPF, denominada GRENJ-LPF. Esse LPF possui features de granularidade fina, visto que se optou pelo Cenário de Utilização 2 para averiguar todo o processo de modularização, conforme comentado anteriormente.

5.4 Considerações Finais

O processo *FMDtoLPF* é constituído por atividades que servem para direcionar a modularização do FMD em features, gerando uma LPF. Como tal processo foi extraído de forma genérica da modularização de um único framework, tais atividades não são definitivas e fixas, uma vez que podem ser adaptadas e estendidas por outros pesquisadores.

Segundo a experiência relatada anteriormente, a utilização de modelos pode ser uma alternativa relevante no contexto de LPFs pela possibilidade do usuário selecionar um conjunto de features que julgar necessárias para o membro e pelo fato do código correspondente ser gerado em tempo de projeto, enquanto que utilizando somente aspectos e a linguagem Java, o desenvolvedor precisa copiar diversos pacotes contendo aspectos, correndo o risco de selecionar algum acidentalmente ou deixar de selecionar os aspectos necessários para o membro que se deseja.

Com aspectos, a modularização de features de granularidade fina pode se tornar muito complexa, apesar de existir vários mecanismos que possibilitam capturas de argumentos, modificação de retornos de métodos, entrecorte de construtores/métodos, além das declarações intertipo. Na maioria das vezes, o desenvolvedor precisa tratar apenas uma linha dentro de um método ou um atributo em um construtor e isso demanda um esforço considerável e tedioso quando se considera um framework de grande porte. Além disso, o número de aspectos a serem criados cresce exponencialmente à medida que as combinações possíveis entre as features de granularidade fina são consideradas. Em contrapartida, a facilidade para modularizar o FMD com templates foi evidente, seja considerando features de granularidade grossa ou fina, pois se deve apenas rotular os trechos de código com as features correspondentes.

Capítulo 6

ESTUDO COMPARATIVO ENTRE FMD E LPFS

Neste capítulo apresenta-se um estudo comparativo entre um FMD e duas LPFs obtidas com a aplicação do processo de modularização proposto. Nesse estudo são utilizadas algumas métricas com o intuito de evidenciar as melhorias obtidas com a modularização, em termos de reuso, arquitetura e manutenção. O FMD considerado foi o GRENJ, assim como no experimento. As LPFs correspondem aos dois cenários de utilização apresentados anteriormente. Uma versão com as features de granularidade grossa, isto é, features em nível de subdomínios e outra de granularidade fina, que expõe as variabilidades existentes.

6.1 Comparação entre um FMD e membros de duas LPFs quanto ao reuso, composabilidade e manutenção

Conforme observado nos capítulos anteriores, a utilização de frameworks no contexto de desenvolvimento de software tornou-se expressiva por conta de mecanismos oferecidos pelo paradigma orientado a objetos, em termos de flexibilidade, produtividade e redução no tempo necessário para o desenvolvimento. Todavia, quando os frameworks passam por evoluções sem um planejamento adequado para agregar novas features, eles tornam-se muitas vezes inflexíveis arquiteturalmente, ocasionando dificuldades relacionadas principalmente com o reuso e a manutenção.

Para comprovar esse efeito e evidenciar que a abordagem de LPF pode ser utilizada nesse contexto, realizou-se comparações envolvendo três aplicações

(hipotéticas) com as seguintes finalidades: Aluguel de Veículos (*Ap.1*), Venda de equipamentos domésticos (*Ap.2*) e Manutenção de motocicletas (*Ap.3*). Cada aplicação foi desenvolvida três vezes. Primeiro com o apoio do GRENJ, aqui chamado de FMD. Em seguida, com o apoio de um membro de uma LPF obtida com a modularização desse FMD, conforme o **Cenário de Utilização 1** (LPF1). Por fim, a terceira versão foi desenvolvida por meio da instanciação de um membro de uma segunda LPF obtida conforme o **Cenário de Utilização 2** (LPF2). Com base nas três versões de cada aplicação, realizamos um estudo comparativo. Vale ressaltar que essas LPFs foram implementadas por meio de templates Acceleo e as aplicações possuem tamanhos diferentes em termos de quantidade de classes e features utilizadas dos frameworks.

Assim, para esclarecer a estrutura dessas aplicações em termos de classes, bem como as principais classes utilizadas do framework, foram elaborados três diagramas de classes que representam os requisitos dessas aplicações. Na Figura 26 ilustra-se o diagrama de classes da Ap.1. Na Figura 27, o diagrama de classes para a Ap.2 e por fim a Figura 28, para a Ap.3.

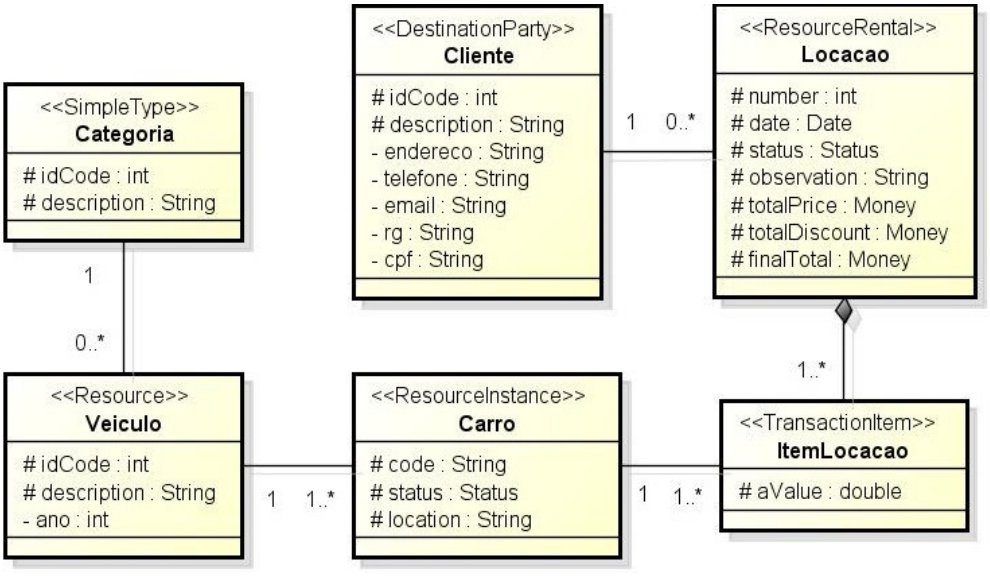


Figura 26: Diagrama de Classes da Ap.1 – "Aluguel de Veículos"

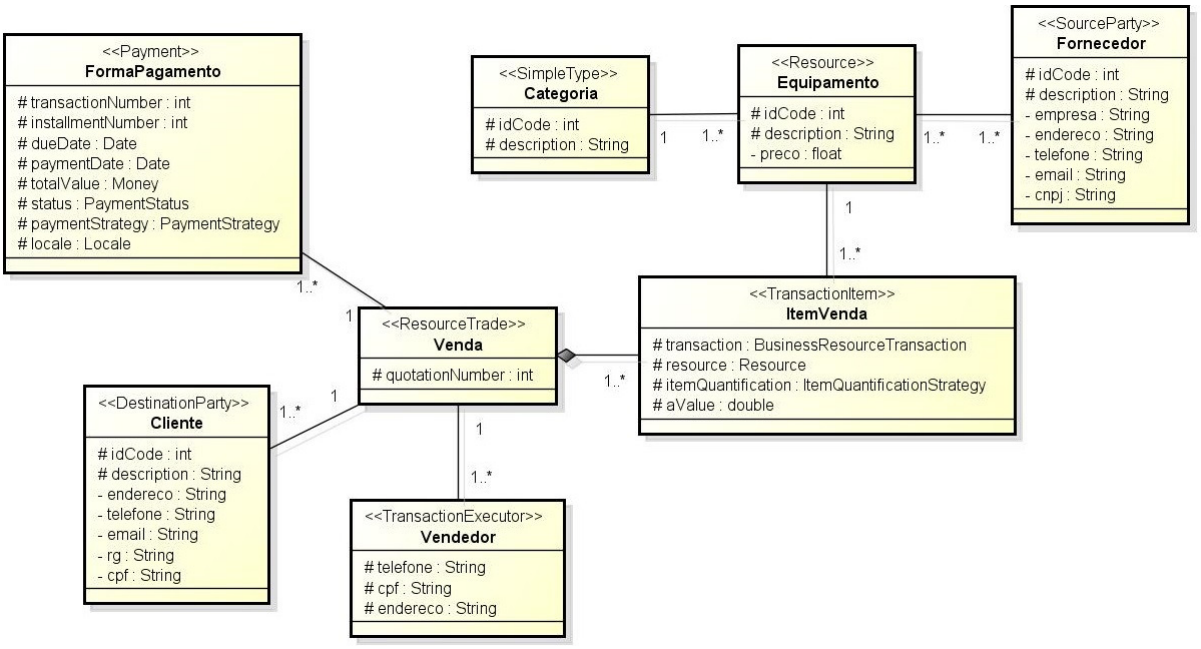


Figura 27: Diagrama de Classes da Ap.2 - "Venda de equipamentos domésticos"

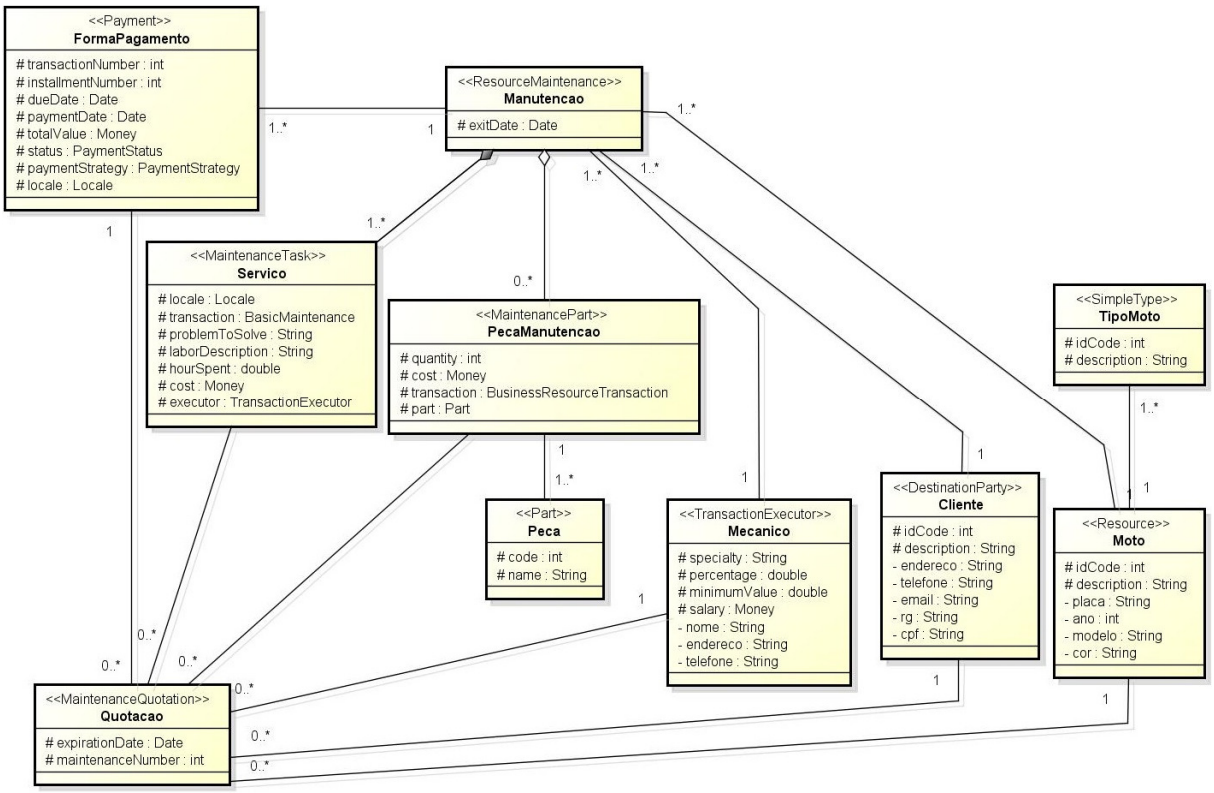


Figura 28: Diagrama de Classes da Ap.3 – "Manutenção de motocicletas"

Observa-se que nos três diagramas de classes, todas as classes possuem um estereótipo. Por exemplo, a classe *Cliente* no diagrama da Figura 26 possui o estereótipo "<<*DestinationParty*>>" e assim por diante. Isso serve para descrever a classe do framework que foi estendida. Os atributos com o prefixo "**#**" indicam que

são provenientes das classes do framework. Vale ressaltar que nas classes dos diagramas não estão todos os atributos provenientes de herança, mas somente os das classes diretamente estendidas. A hierarquia de herança das classes do framework pode ser visualizada no Anexo B.

Na Figura 29 ilustra-se o diagrama de features que representa a LPF obtida com a modularização conforme o **Cenário de Utilização 1,** sendo o mesmo ilustrado na parte (c) da Figura 19.

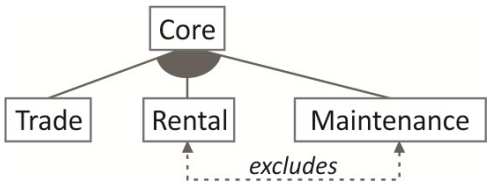


Figura 29: LPF segundo o Cenário de Utilização 1 (parte(c) da Fig. 19)

Considerando esse diagrama, os membros dessa LPF considerados no estudo foram: "*Core* + *Rental*" (C+R) para a Ap.1, "*Core* + *Trade*" (C+T) para a Ap.2 e "*Core* + *Maintenance*" (C+M) para a Ap.3, enquanto que o diagrama de features referente à LPF obtida conforme o **Cenário de Utilização 2** e as configurações para as mesmas aplicações são ilustradas na Figura 30. Observa-se que os membros dessa segunda LPF, puderam ser mais direcionados aos requisitos das aplicações, como pode ser visualizado pelas features marcadas com "√". Por exemplo, as features que compõe o membro da LPF2 para a Ap.1 são: *Core, Resource, ResourceQuantification, BusinessTransaction, Simple, Nested, Instantiable, Rental, TransactionItemized* e *DestinationParty*. Considerando essas aplicações, o estudo comparativo foi conduzido com a utilização de algumas métricas:

- Número total de linhas de código (*KLOC*) do framework;
- Número total de métodos abstratos (*hot spots*) existentes;
- Quantidade de métodos concretizados imprescindíveis para tornar a aplicação funcional;
- Número de linhas dos métodos concretizados;
- Quantidade de parâmetros dos métodos abstratos;
- Número total de features disponíveis no framework;
- Número de features utilizadas para a aplicação;
- Número de variabilidades/features que podem ser utilizadas futuramente considerando o domínio da aplicação;
- Número de variabilidades/features que dificilmente serão utilizadas.

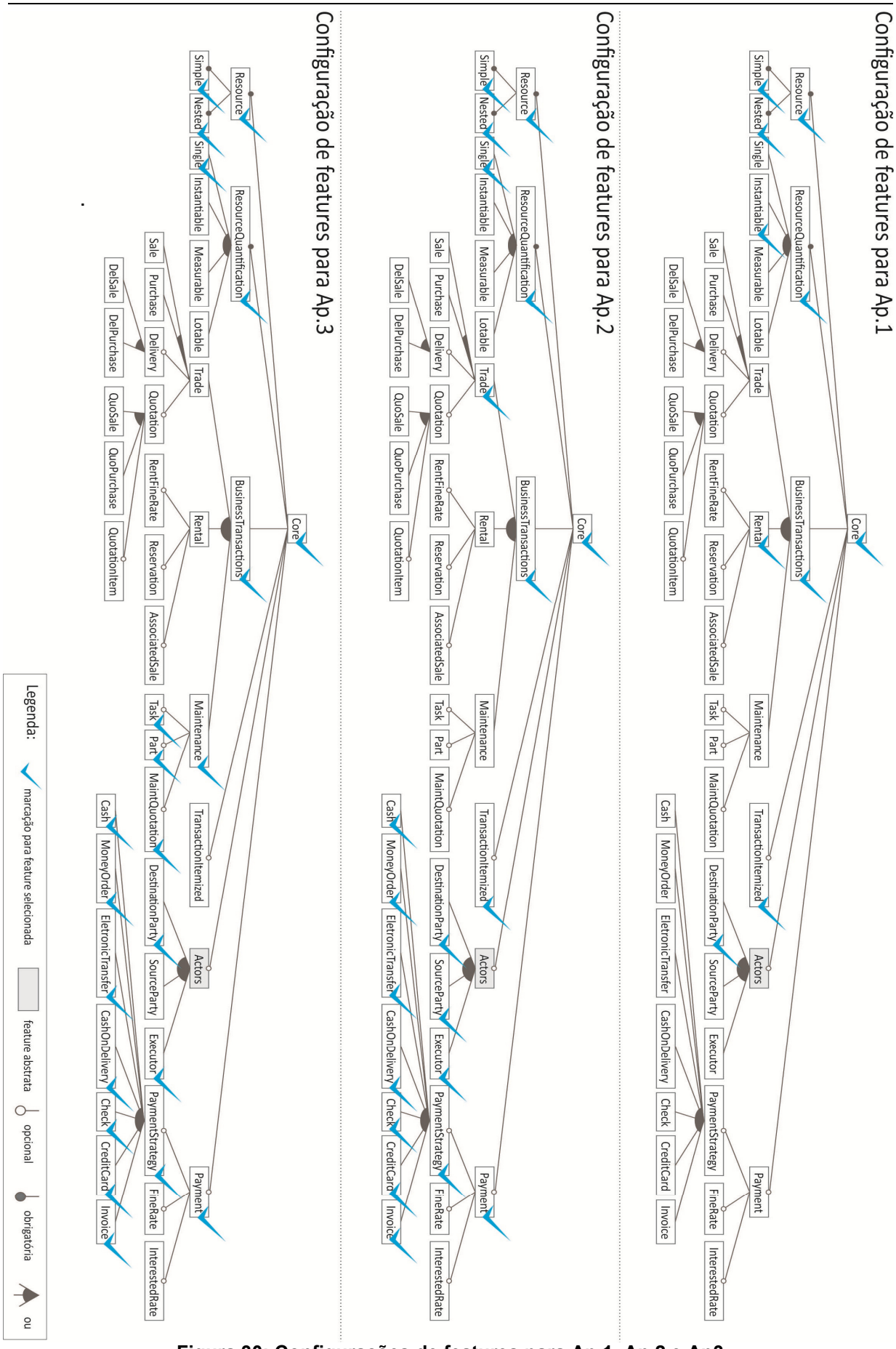


Figura 30: Configurações de features para Ap.1, Ap.2 e Ap3

No Quadro 12 são descritos os dados obtidos com a aplicação dessas métricas nas três versões de cada aplicação. A arquitetura final das aplicações desenvolvidas com o apoio do FMD resultou no código base da aplicação com a estrutura completa do framework, incluindo partes desnecessárias que não seriam utilizadas. As mesmas aplicações ao serem desenvolvidas com um membro da LPF1, incluíram somente a parte correspondente aos seus domínios, o que inclui variabilidades não utilizadas, mas todas são possíveis em evoluções futuras. Posteriormente, as aplicações foram desenvolvidas com membros da LPF2, utilizando um conjunto mais restrito de features do que os membros da LPF1 conforme os requisitos dessas aplicações.

	FMD (GRENJ)	Membro da LPF1	Membro da LPF2	FMD (GRENJ)	Membro da LPF1	Membro da LPF2	FMD (GRENJ)	Membro da LPF1	Membro da LPF2
Métricas	*Aluguel de Veículos (Ap.1)*			*Venda de equipamentos domésticos (Ap.2)*			*Manutenção de motocicletas (Ap.3)*		
Número total de linhas de código (*KLOC*) do framework	*8,5*	*4,2*	*3,4*	*8,5*	*5,6*	*4,1*	*8,5*	*3,5*	*2,9*
Número total de métodos abstratos (*hot spots*) existentes	*158*	*36*	*16*	*158*	*56*	*28*	*158*	*62*	*29*
Número de métodos concretizados para tornar a aplicação funcional	*20*	*20*	*16*	*48*	*48*	*28*	*55*	*55*	*29*
Número de linhas dos métodos concretizados	*52*	*52*	*40*	*187*	*187*	*132*	*89*	*89*	*63*
Número de parâmetros dos métodos concretizados	*12*	*12*	*8*	*19*	*19*	*16*	*18*	*18*	*14*
Número total de variabilidades/features disponíveis do framework	*43*	***C+R*** *C=25* *R=4* ***29***	*10*	*43*	***C+T*** *C=25* *T=10* ***35***	*21*	*43*	***C+M*** *C=25* *M=4* ***29***	*22*
Número de variabilidades/features utilizadas pela aplicação	*43/10*	*29/10*	*10/10*	*43/21*	*35/21*	*21/21*	*43/22*	*29/22*	*22/22*
Número de variabilidades/features que podem ser utilizadas futuramente considerando o domínio da aplicação (sem considerar as já utilizadas)	***18***	*C=16* *R=2* ***18***	*0*	***14***	*C=5* *T=9* ***14***	*0*	***7***	*C=7* *M=0* ***7***	*0*
Número de variabilidades/features que dificilmente serão utilizadas (referem-se as features de outros domínios)	*15*	*0*	*0*	*8*	*0*	*0*	*14*	*0*	*0*

Quadro 12: Métricas utilizadas nas aplicações desenvolvidas

No Quadro 12, o número total de linhas de código (*KLOC*) foi calculado utilizando o *plug-in Eclipse Metrics*[13] e as demais métricas por observação direta do código. O FMD (GRENJ) com 8,5 *KLOC* e 158 *hot spots*, ao ser instanciado pelas três aplicações, é considerando como uma parte monolítica e indivisível. Desse modo, todas as variabilidades do FMD tornam-se presentes nas aplicações mesmo que algumas nunca sejam utilizadas. Isso não acontece ao utilizar um membro das LPFs, visto que há uma redução da quantidade de código e consequentemente da quantidade de *hot spots* que são disponibilizados por seus membros. Por exemplo, no membro da LPF1 utilizado pela Ap.1 havia 4,2 *KLOC* e no membro da LPF2 utilizado pela mesma aplicação continha 3,4 *KLOC* e assim por diante. Isso significa uma redução de 50,58% e 60% de código do framework nessa aplicação.

Com relação aos *hot spots* fornecidos pelos frameworks provenientes da LPF1 e LPF2 para a Ap.1, a redução foi de 77,22% e 89,88%, respectivamente. Essa redução também ocorre nos frameworks utilizados pelas demais aplicações. No desenvolvimento das três aplicações com o apoio do FMD e de membros da LPF1, foi necessário concretizar métodos que não necessariamente estavam ligados aos requisitos dessas aplicações. Isso acontece com o reuso do FMD, pois nesse processo todas as suas variabilidades fornecidas tornam-se parte das aplicações. Quanto aos membros da LPF1, a feature *Core* inclui todas as variabilidades externas aos subdomínios: *Trade*, *Rental* e *Maintenance*.

Desse modo, na LPF1 não há possibilidade de selecionar uma parte do *Core*, o que poderia contribuir para reduzir a quantidade desses métodos. Conforme os dados obtidos na terceira métrica, para a Ap.1 foi necessário concretizar 20 métodos, para a Ap.2, 48 métodos e para Ap.3, 55 métodos. Porém, ao desenvolver as aplicações com o apoio de membros da LPF2, foi possível instanciar somente os métodos relacionados aos requisitos das aplicações, reduzindo assim o esforço durante a instanciação do framework. Isso representa uma redução de 20% de métodos a serem concretizados para Ap.1, 41,66% para Ap.2 e 47,27% para Ap.3.

Os dados obtidos com a quarta e quinta métrica indicam que o número de linhas dos métodos concretizados e seus parâmetros também continuaram constantes na utilização do FMD e dos membros da LPF1. Porém, quando as aplicações foram desenvolvidas com membros da LPF2, houve redução dos dados

[13] http://www.sourceforge.net/projects/metrics

obtidos nessas duas métricas. Com relação ao número de linhas, para a Ap.1 a redução foi de 52 para 40 linhas, representando 23%; para Ap.2 foi de 187 para 132 linhas, 29,41% e para Ap.3 foi de 89 para 63 linhas de código, representando 29,21%. Para esclarecer, ao instanciar um membro da LPF2 gerando a Ap.1, por exemplo, foi possível utilizar um conjunto restrito de variabilidades e concretizar métodos ligados aos requisitos dessa aplicação, cobertos pelo subdomínio *Rental*, sem precisar concretizar certos métodos referentes à features desnecessárias com um "retorno nulo" para indicar ao framework que uma determinada funcionalidade não seria utilizada. De modo geral, a inexistência desses métodos nos membros da LPF2 afeta diretamente a quantidade de parâmetros.

Para calcular os valores das quatro últimas métricas que estão relacionadas com variabilidades/features fornecidas pelos frameworks às três aplicações, foi necessário admitir um nível de equivalência. Para isso, adotou-se como quantidade total de features, as que são fornecidas pela LPF2 (ilustrado na Figura 30). A partir disso, para os membros da LPF1, consideraram-se as features inclusas "internamente" em: *Core, Rental, Trade* e *Maintenance.* Para o FMD, mesmo não estando modularizado em features, foi considerado fornecendo um total de 43 variabilidades.

Por exemplo, o membro da LPF1 utilizado pela Ap.2, fornece 35 features que estão inclusas em apenas duas: *Core e Trade*. No *Core*, existem 24 features e em *Trade* outras 9, totalizando 35 e não 33 pois as features *Core* e *Trade* não são abstratas. Por isso, a linha do Quadro 12 correspondente a métrica "Número total de variabilidades/features disponíveis do framework" para essa aplicação contém C=25 e T=10, enquanto que o membro da LPF2 para a Ap.2 fornece apenas 21 features com base nos requisitos dessa aplicação. Essa mesma análise também é feita para Ap.1 e para Ap.3.

Para obter os valores referentes a métrica "Número de variabilidades/features utilizadas pela aplicação", foi necessário considerar o valor obtido na métrica anterior. Para a Ap.1, por exemplo, apesar de o FMD fornecer 43 variabilidades, a aplicação exigiu somente 10. Por isso, a descrição "43/10". O membro da LPF1 forneceu 29 features. Dentre essas, 10 foram utilizadas e as demais se referem à features que pertencem ao subdomínio *Rental* e podem ser empregadas futuramente. O membro da LPF2 forneceu um conjunto ainda mais restrito, somente

as 10 features necessárias à aplicação. Para as demais aplicações, o cálculo foi o mesmo.

Com a métrica “Número de variabilidades/features que podem ser utilizadas futuramente considerando o domínio da aplicação” pode-se observar a quantidade de features que sobraram no reuso dos frameworks e são prováveis de serem usadas no futuro. Vale ressaltar que nessa métrica considerou-se que a evolução das três aplicações não implicaria em agregar outro domínio. Por exemplo, uma aplicação do domínio *Trade* evoluir agregando *Rental*. Considerando essa restrição, 18 variabilidades que não foram utilizadas pela Ap.1 são possíveis futuramente. Esse valor é dado pela soma das features remanescentes que ainda podem ser agregadas ao membro utilizado, caso houver necessidade. No diagrama de features que representa a LPF2 há uma feature chamada *AssociatedSale* pertencente a *Rental.* A seleção da feature *AssociatedSale* implica na inclusão da feature *Sale* no membro. Isso quebra a restrição mencionada anteriormente no cálculo. As 18 features remanescentes que são possíveis de serem empregadas na Ap.1 podem ser decompostas da seguinte forma: 16 pertencem a *Core* e apenas duas a *Rental (C=16, R=2)*. Essas duas features em *Rental* são: *RentalFineRate* e *Reservation*, ou seja, *AssociatedSale* não foi considerada. Com relação ao membro da LPF2 não há features remanescentes, visto que os membros da LPF2 são criados com essa intenção. Para Ap.2 e Ap.3, tem-se a mesma ocorrência.

Observa-se na última linha do Quadro 12, a quantidade de features fornecidas pelo FMD que dificilmente serão empregadas nas aplicações, mesmo que sofram diversas evoluções. Geralmente, essas features são específicas de subdomínios diferentes de determinados conjuntos de aplicações. Com a utilização do FMD para desenvolver a Ap.1, 15 das 43 features, dificilmente serão utilizadas e mesmo assim continuam na aplicação. Isso representa 34,88% de features dispensáveis. Para a Ap.2, essa quantidade foi de 18,60% e para a Ap.3, 32,55%. Observa-se que nessas aplicações, ao utilizar membros da LPF1 ou da LPF2, não haviam features que pertencessem a domínios diferentes dos domínios dessas aplicações. Note-se que o GRENJ é um framework relativamente pequeno. Desse modo, acredita-se que se esse tipo de análise fosse conduzido em frameworks maiores, a quantidade de features desnecessárias poderia ser mais expressiva.

Considerando as aplicações, os frameworks e os dados obtidos com as métricas, existem algumas relações interessantes. Com base nos diagramas de classes, o tamanho das aplicações é variado, isto é, a quantidade de classes de cada aplicação é diferente. A Ap.1 possui 6 classes, Ap.2 8 classes e a Ap.3 10 classes. Isso implicou diretamente na quantidade de features utilizadas, bem como na quantidade de *hot spots* disponibilizados pelos frameworks. Na Ap.1, a redução do número de *hot spots*, comparando o FMD com o membro da LPF1 foi de 77,22%, enquanto que a porcentagem na Ap.2 foi 64,56% e para Ap.3, 60,76%. Considerando os membros da LPF2 essa redução foi ampliada, 89,88% no membro utilizado na Ap.1, 82,28% na Ap.2 e 81,65% para Ap.3.

Por um lado, uma LPF obtida segundo a ideia do cenário de utilização 1, serve para atender demandas por frameworks que comportam todas as features de um determinado subdomínio. Logo, membros dessa LPF são direcionados ao desenvolvimento de aplicações com chances plausíveis de utilizar todas as features fornecidas. Nesse caso, todas as features comuns aos subdomínios são inclusas nesses membros, isto é, não há possibilidade desses membros conterem uma parcela dessas features. Desse modo, deve-se considerar também se as aplicações que se deseja desenvolver a partir desses membros, poderão requerer essa parte comum de forma completa, seja no desenvolvimento ou em futuras evoluções. De outro lado, uma LPF obtida segundo a ideia do cenário de utilização 2, serve para atender demandas particulares, ou seja, além de permitir a seleção de um conjunto restrito de features que pertencem a um subdomínio, essa LPF também permite a seleção de uma parcela das features comuns aos subdomínios, ou seja, features que não possuem um subdomínio específico.

De modo geral, o processo de instanciação de um framework menor e direcionado ao domínio de uma aplicação pode ser mais fácil, produtivo e principalmente menos tendencioso a erros, quando comparado com a versão inteira e extensa do mesmo framework. Afinal, quanto mais variabilidades são disponibilizadas e essas não tenham sido implementadas com um gerenciamento adequado, maiores podem ser as chances de um EA cometer falhas.

Como ilustrado nos diagramas das Figuras 29 e 30, a partir de uma LPF pode-se compor features e gerar versões alternativas, sejam frameworks direcionados aos domínios das aplicações ou mais restritos, considerando requisitos específicos. Assim, a flexibilidade arquitetural fornecida pelas LPFs permitem

melhores níveis de composição. Em um FMD isso não seria possível, pois não existem tais possibilidades. Além disso, com um framework extenso torna-se complexo expor para os usuários quais são as funcionalidades disponíveis e como estão organizadas. Com o emprego de features, há maior separação de interesses e com isso uma arquitetura mais clara (Batory et al., 2002). Esse é um fator que pode facilitar as atividades de evolução/manutenção quando comparado com o FMD original, quanto ao gerenciamento de variabilidades.

6.2 Considerações Finais

Neste capítulo foi apresentado um estudo comparativo envolvendo membros de duas LPFs, obtidas segundo a ideia do Cenário de Utilização 1 e do Cenário de Utilização 2, contra a versão do FMD original. Nesse estudo, foram desenvolvidas três aplicações de domínios distintos. Cada aplicação foi desenvolvida com o FMD e depois com membros das LPFs. A partir disso, foram utilizadas algumas métricas e os resultados obtidos foram favoráveis e vão ao encontro das expectativas iniciais do processo desenvolvido que incluem redução dos esforços, aumentando a produtividade.

Capítulo 7

EXPERIMENTO PARA AVALIAR O ESFORÇO NA MODULARIZAÇÃO DOS SUBDOMÍNIOS DO GRENJ COM POA E MDD

Neste capítulo são apresentados os detalhes de um experimento que foi conduzido com o intuito de averiguar o impacto das técnicas POA e MDD no processo de modularização dos subdomínios de um FMD, visando obter LPFs no final desse processo.

7.1 Introdução ao experimento

O processo de avaliação de software é uma atividade essencial para melhorar o desenvolvimento de software em uma organização (Ares et al., 1998). Esse processo oferece apoio sistemático, disciplinado, quantificável e controlado sob a perspectiva da atividade humana em um processo de criação de software (Travassos et al., 2002). Por meio da avaliação, pode-se verificar a efetividade de novas abordagens, ferramentas, processos e estratégias de desenvolvimento de software em comparação com as existentes.

A avaliação experimental em Engenharia de Software, segundo Wohlin et al. (2000), envolve uma sequência de etapas, como a definição, planejamento, a condução do experimento e por fim, a análise dos resultados obtidos. Uma das vantagens ao realizar um experimento é o controle que se tem dos participantes,

objetos e instrumentação. Além disso, podem-se desenvolver análises estatísticas por meio de métodos de teste de hipóteses, observando as conclusões do experimento e as possibilidades de replicação.

Nesse sentido, realizou-se uma avaliação controlada com a aplicação da metodologia experimental (Wohlin et al., 2000), com o intuito de averiguar o impacto da utilização das técnicas Orientação a Aspectos com a linguagem AspectJ e MDD com templates Acceleo, na modularização dos subdomínio de um FMD em features, no sentido de obter uma LPF no final do processo. Acredita-se que os dados obtidos nesse experimento possam servir como indicativo para EFs na adoção de uma dessas tecnologias para reestruturar FMDs.

O processo para modularizar subdomínios de um FMD, conforme comentado anteriormente, requer atividades de alto nível, como: (a) identificar os subdomínios do FMD, (b) identificar os trechos de código relacionados a esses subdomínios e (c) modularizar o FMD de acordo com um diagrama de features que representa a disposição dos subdomínios. Considerando a complexidade para realizar tais atividades, decidiu-se avaliar apenas a terceira atividade. Assim, previamente identificaram-se os subdomínios do FMD e forneceu-se um diagrama de features para os participantes. Como o FMD utilizado neste experimento foi o GRENJ, considerado na elaboração do processo de modularização, comentado anteriormente, o diagrama de features disponibilizado é ilustrado na parte (c) da Figura 19.

Este capítulo está organizado da seguinte forma: a Seção 7.1.1 apresenta a Definição do Estudo, contendo o objetivo, o objeto do estudo, o enfoque qualitativo/quantitativo e a perspectiva desse experimento; a Seção 7.1.2 descreve o Planejamento do Experimento, incluindo a discussão acerca da seleção do contexto, formulação das hipóteses, variáveis, seleção dos participantes, projeto do experimento, instrumentação e avaliação da validade; a Seção 7.1.3 descreve o desenvolvimento da experimentação, incluindo a preparação, execução e validação dos dados obtidos; a Seção 7.1.4 apresenta a Análise e Interpretação dos dados alcançados, por meio de um detalhamento estatístico, redução de dados, além do teste das hipóteses e por fim, na Seção 7.1.5 visualizam-se as Conclusões desse experimento.

7.1.1 Definição

7.1.1.1 Objetivo

O objetivo do estudo foi averiguar qual técnica de implementação, AspectJ ou Templates Acceleo, exige menos esforço para se modularizar um FMD em uma LPF. O esforço foi medido em termos de tempo (produtividade), ocorrência de erros estruturais e de inconsistência (qualidade). Os erros estruturais considerados no experimento referem-se às dependências por métodos/atributos pertencentes às classes que foram modularizadas de maneira incorreta, ao invés de erros ligados aos caminhos lógicos no software, decorrentes de Testes Estruturais (Pressman, 2006), enquanto que os erros de inconsistência, referem-se às implementações de regras de geração/composição de código que resultam em introduções de métodos e atributos em classes indevidas.

7.1.1.2 Sujeito do Estudo

Duas técnicas de desenvolvimento foram consideradas para a modularização dos subdomínios de um FMD: POA com a linguagem AspectJ e MDD com templates Acceleo.

7.1.1.3 Enfoque Quantitativo

O enfoque quantitativo considera a identificação do esforço para a modularização de um framework em uma LPF, com base nos tempos despendidos e nas ocorrências de erros estruturais e de inconsistência durante o processo de modularização.

7.1.1.4 Enfoque Qualitativo

O enfoque qualitativo é usado para determinar a técnica que exige menos esforço.

7.1.1.5 Perspectiva

O experimento foi conduzido sob a perspectiva do Engenheiro de Framework que possui interesse em modularizar FMD em LPFs para atender demandas por

frameworks mais direcionados às necessidades das aplicações e evitar a presença de features desnecessárias do framework na aplicação final.

7.1.1.6 Objetos de Estudo

Os objetos de estudo neste experimento consistem na produtividade em termos de tempo necessário para concluir o processo de modularização de um FMD em uma LPF e a ocorrência de erros estruturais e de inconsistência durante esse processo.

7.1.2 Planejamento

O experimento foi planejado de forma a satisfazer as seguintes questões de pesquisa: “Há diferença em termos de produtividade entre utilizar POA ou MDD para modularizar um FMD?” **(QP1)**, e “Existe diferença em termos de erros estruturais e de inconsistência?” **(QP2)**.

As questões de pesquisa foram elaboradas com o objetivo de comparar as duas técnicas no processo de modularização. Assim, o experimento não foi planejado para mostrar que uma LPF é melhor que seu FMD precursor.

Para responder a primeira questão, nós coletamos e avaliamos o tempo despendido para realizar a modularização. Similarmente, para responder a segunda questão, analisamos um formulário que os sujeitos informaram os erros que foram encontrados. Dessa maneira, o tempo total inclui o tempo despendido também para tratar os erros. A fase de planejamento foi dividida em seis partes, que são descritas nas próximas subseções.

7.1.2.1 Seleção do Contexto

O experimento ocorreu em ambiente universitário, sendo realizado com estudantes de pós-graduação no Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos (UFSCar), no âmbito da disciplina “Tópicos em Engenharia de Software 2012”.

7.1.2.2 Formulação de Hipóteses

Um experimento geralmente é formulado por meio de hipóteses (Travassos et al., 2002; Wohlin et al., 2000). A hipótese principal chama-se hipótese nula e declara que não há nenhum relacionamento estatisticamente significante entre a causa e efeito (Travassos et al., 2002). O ideal no experimento é rejeitar a hipótese nula a favor de uma ou algumas hipóteses alternativas, com base em teste estatístico. As hipóteses são declaradas formalmente e os dados coletados durante a condução do experimento servem para afirmar ou rejeitar certas hipóteses alternativas.

7.1.2.2.1 Demonstração das Hipóteses

A **QP1** foi formalizada da seguinte forma: a **hipótese nula (H_0)** é verdadeira caso o esforço necessário em termos de tempo para modularizar o FMD utilizando as técnicas POA e MDD forem equivalentes estatisticamente e A **hipótese alternativa (H_1)** é verdadeira caso exista diferença entre o esforço necessário em termos de tempo, assim as técnicas não são equivalentes. As hipóteses para QP1 podem ser formalizadas como:

$$H_0: \mu POA = \mu MDD \qquad H_1: \mu POA \neq \mu MDD$$

Similarmente, sob enfoque quantitativo, a **QP2** foi formalizada da seguinte forma: a **hipótese nula (H_0)** é verdadeira caso a quantidade de erros no processo de modularização do FMD para LPF com as duas técnicas descritas anteriormente sejam equivalentes estatisticamente. A **hipótese alternativa (H_1)** é verdadeira caso exista diferença entre a quantidade de erros. Assim as técnicas não são equivalentes. As hipóteses para QP2 podem ser formalizadas como:

$$H_0: \mu POA = \mu MDD \qquad H_1: \mu POA \neq \mu MDD$$

7.1.2.3 Seleção das Variáveis

As variáveis dependentes são: “tempo despendido para modularizar um FMD para uma LPF” e o “número de erros encontrados durante essa modularização”. As variáveis independentes são:

- **LPF:** Solicitamos aos participantes para criar duas LPFs a partir da modularização de um FMD. A única diferença entre elas foi a técnica usada (POA ou MDD);
- **Os subdomínios *Trade* e *Rental* do GRENJ**;
- **Ambiente de Desenvolvimento:** Eclipse 4.2.1;
- **Tecnologias:** Java version 6, AspectJ 1.7.1, Acceleo 3.3.2 e FeatureIDE;
- ***Plug-in* desenvolvido para a transformação do modelo de features.**

7.1.2.4 Seleção dos Participantes

A seleção dos participantes foi realizada por meio de uma amostragem não probabilística por conveniência (Wohlin et al., 2000). Essa seleção envolveu a participação de 14 alunos do 1º ano de pós-graduação em Ciência da Computação da UFSCar, no âmbito da disciplina: “Tópicos em Engenharia de Software - 2012”. Nas seções a seguir, chamamos os participantes pelo termo “sujeitos”.

7.1.2.5 Projeto do Experimento

O projeto do experimento descreve como os testes experimentais são organizados e executados. O experimento seguiu o princípio da separação dos participantes em blocos homogêneos (Wohlin et al., 2000). Assim, foi possível evitar o impacto do nível de experiência nos resultados quanto ao fator *técnica de modularização,* aumentando dessa forma a precisão do experimento.

Para dividir os sujeitos em grupos balanceados (homogêneos), em primeiro lugar, solicitou-se que eles preenchessem um *Formulário de Categorização* quanto ao nível de experiência levando em conta os requisitos do experimento. Em seguida, aplicaram-se exercícios relacionados ao experimento para verificar de forma segura, se suas respostas do formulário eram de fato verdadeiras. Assim, baseando-se nesses dados, os sujeitos foram divididos em dois grupos de 7 integrantes. Esses grupos foram submetidos a um experimento piloto e com os dados coletados, os grupos foram novamente rebalanceados para o experimento real.

O *Formulário de Categorização* que foi usado contempla questões de conhecimento quanto: a linguagem Java, AspectJ, Acceleo, experiência em programação, GRENJ, GRN e a IDE Eclipse. Na Figura 26 é ilustrado o resultado da

aplicação desse formulário em um gráfico de barras. As barras representam os níveis de experiência de cada sujeito e os retângulos sobre as barras contém um valor interno, que representa suas médias aritméticas ponderadas. Essas médias foram calculadas segundo a equação: $Mp = (10*x1 + 10*x2 + 10*x3 + 20*x4 + 30*x5 + 20*x6+x7)/100$. Em que *x1* representa o nível de experiência com a linguagem Java, *x2* com AspectJ, *x3* com templates Acceleo, *x4* em programação, *x5* com o GRENJ, *x6* com a GRN e *x7* com a IDE Eclipse. Os níveis estão organizados no intervalo de 1 a 3: o nível 1 representa o nível básico, o nível 2 representa o nível intermediário e o nível 3, o nível avançado de experiência. Os sujeitos são representados por meio do prefixo "S" combinado com um número inteiro de 1 até 14. Vale ressaltar que considerando os dados obtidos com a aplicação desse formulário, exercícios e os dados coletados no experimento piloto, os sujeitos foram separados de forma balanceada em dois grupos. Os sujeitos S1 a S7 pertencem ao grupo 1 e os sujeitos S8 a S14 compõem o grupo 2.

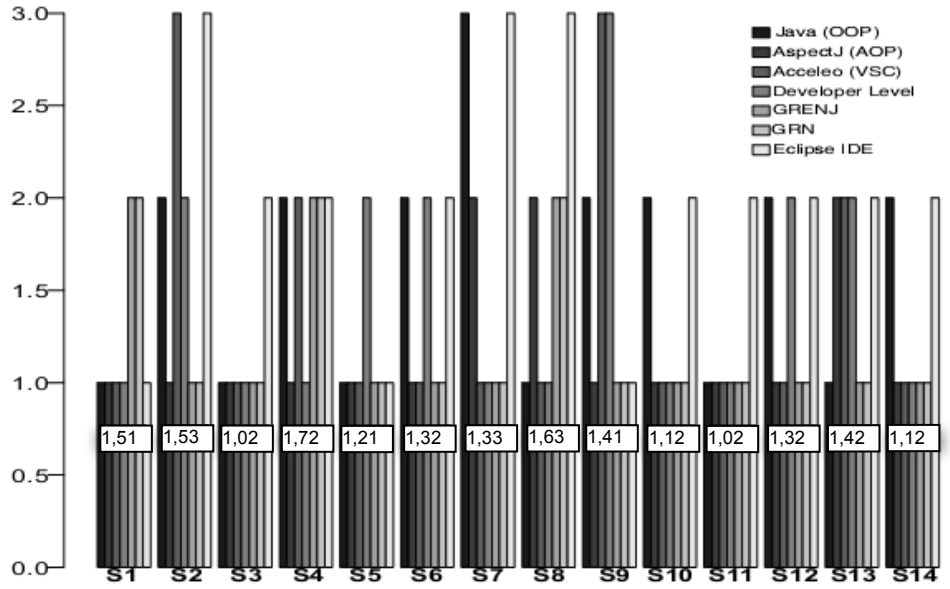


Figura 31: Níveis de experiência individuais dos sujeitos

No Quadro 13 é descrita a configuração do experimento. Na etapa de Treinamento, os sujeitos foram treinados em POA utilizando AspectJ e MDD, com Acceleo. Além disso, foram treinados em como utilizar essas tecnologias para modularizar os subdomínios *Rental* e *Trade* do GRENJ, no sentido de obter "LPFs parciais", i.e., os sujeitos não tiveram que reestruturar o GRENJ para obter uma LPF

completa, mas apenas uma LPF que permite gerar membros de acordo com o subdomínio modularizado.

As etapas do experimento piloto e real possuíam as seguintes atividades: “Modularizar o subdomínio *Trade*” e “Modularizar o subdomínio *Rental”,* utilizando POA ou MDD. O resultado de cada modularização é uma LPF parcial que permite gerar um membro com todas as funcionalidades do subdomínio modularizado, evitando a presença de features pertencentes a outros subdomínios. Por exemplo, a modularização do subdomínio *Rental* resulta em uma LPF parcial que permite gerar um framework sem as features *Trade* e *Maintenance*. Para testar esse membro, forneceu-se um *workspace* com aplicações prontas para utilizar os membros. Assim, os sujeitos adicionavam um membro da LPF obtida às aplicações e em seguida executavam o caso de teste.

Vale enfatizar que o procedimento realizado com POA e MDD, apesar de serem parecidos, envolviam algumas diferenças. Por exemplo, para obter um membro da LPF obtida com AspectJ, cada sujeito deveria tratar as classes que colaboravam com a implementação do subdomínio a ser modularizado por meio de aspectos. Feito isso, todas as classes que implementavam esse subdomínio e os aspectos criados tinham que ser movidos para um segundo projeto dentro do *workspace.* Esse projeto temporário tinha a única função de servir como repositório, evitando que as classes/aspectos fossem removidas. Em seguida, o framework resultante, sem a parte do subdomínio modularizado era copiado para a aplicação disponibilizada. Por fim, executava-se o caso de teste, se não houvesse erros, isso indicava que o subdomínio modularizado havia se tornado opcional, por exemplo.

No caso da LPF obtida com MDD, especificamente com templates Acceleo, os sujeitos modificaram os templates que geravam as classes afetadas pelo subdomínio modularizado, adicionando novas regras de composição por meio de *tags*. Isso servia para indicar quais trechos do código implementavam determinada feature. Em seguida, os sujeitos usaram um modelo do FeatureIDE (conforme a parte (c) da Figura 19) em que as features diferentes do subdomínio modularizado estavam selecionadas. A partir desse modelo, era gerada uma instância de um metamodelo pelo *plug-in* que foi desenvolvido, conforme comentado anteriormente. Essa instância servia de entrada para o Acceleo. Com base nas features selecionadas e nos templates modificados pelos sujeitos, todas as classes do

membro eram geradas. Posteriormente, bastava adicionar esse membro à aplicação fornecida e executar o caso de teste.

Apesar das atividades, tanto no piloto quanto no experimento real, possuírem as mesmas descrições, i.e., a modularização dos subdomínios *Rental* e *Trade*; essas não foram exatamente iguais, pois isso comprometeria a validade dos dados obtidos no experimento real. Assim, para cada atividade, foram fornecidas diferentes versões dos subdomínios *Trade* e *Rental*. Para indicar essa diferença, no experimento piloto há um "*", que representa a modularização de uma versão diferente dos subdomínios utilizados no experimento real.

O esforço em relação ao tamanho e complexidade para modularizar ambos os subdomínios dessas versões foram equivalentes. Por exemplo, tanto a modularização do subdomínio *Trade* quanto a modularização do subdomínio *Rental* exigiu o tratamento de 6 classes completas e 17 métodos espalhados em outras 7 classes. Para esclarecer, a atividade "Modularizar o subdomínio *Trade*" no piloto demandou que 17 métodos espalhados em 7 classes e 6 classes inteiras fossem modularizadas. Essa atividade no experimento real exigiu o mesmo esforço, mas com outros métodos e classes. O mesmo é válido para as atividades de modularização do subdomínio *Rental*.

	Fase	Grupo 1	Grupo 2
Treinamento	***1ª Fase***	Técnicas POA e MDD	
	2ª Fase	Modularização de subdomínios de um FMD no sentido de obter LPFs utilizado: POA e MDD	
Experimento Piloto	***1ª Fase***	MDD	POA
		*Modularização do Subdomínio Trade	*Modularização do Subdomínio Rental
	2ª Fase	POA	MDD
		*Modularização do Subdomínio Rental	*Modularização do Subdomínio Trade
Experimento Real	***1ª Fase***	MDD	POA
		Modularização do Subdomínio Trade	Modularização do Subdomínio Rental
	2ª Fase	POA	MDD
		Modularização do Subdomínio Rental	Modularização do Subdomínio Trade

Quadro 13: Projeto do Experimento

7.1.2.6 Instrumentação

Para auxiliar os sujeitos no processo de modularização, forneceu-se um mapeamento entre classes e features do GRENJ, simulando que eles possuíam conhecimento completo do framework. Desse modo, tornou-se simples a identificação de quais classes colaboravam com a implementação de quais features. No Quadro 14 é descrita parte desse mapeamento. As linhas representam as features e as colunas, as classes do framework. Cada célula marcada com "X" indica que a classe colabora com a implementação de uma dada feature. Por exemplo, *AbstractCalculator* é uma classe que contribui para a implementação da feature *Core*, *BasicDelivery* contribui com e feature *Trade* e assim por diante. Além disso, também se inseriu comentários no código fonte das classes para indicar quais trechos de código estão relacionados com as features. Isso foi feito para simular que os sujeitos conheciam os lugares que deveriam ser tratados. Em adição, também fornecemos a documentação das classes do GRENJ e um guia com as etapas que deveriam ser seguidas para a modularização.

Classes / Features	AbstractCalculator	BasicDelivery	BasicMaintenance	...
Core	X			...
Trade		X		...
Rental				...
Maintenance			X	...

Quadro 14: Mapeamento entre as classes e as features do GRENJ

Assim, com o modelo de features e o mapeamento, os sujeitos tiveram que obter duas LPFs, cada com uma técnica diferente, mas equivalente em termos de funcionalidade, complexidade e alternativas de composição. Conforme comentado anteriormente, a utilização de templates envolve a criação de regras de composição, mas apenas isso não é o suficiente para a geração de código dos membros de uma LPF, a partir de um subconjunto de features selecionadas. O mecanismo responsável para gerar o código é o Acceleo, mas para essa geração é necessário um modelo como entrada. Assim, foi fornecido um *plug-in* para transformar configurações válidas de um modelo de features do FeatureIDE em uma instância (.xmi) de um metamodelo que também foi criado, conforme comentado

anteriormente. Essa instância serve de entrada para a *engine* do Acceleo gerar código com base nos templates. Dessa forma, a partir da seleção de features de um modelo FeatureIDE, geram-se as classes referentes ao framework requerido com somente os métodos e refinamentos necessários.

7.1.3 Execução

Uma vez que o experimento foi definido e planejado, ele foi executado de acordo com as seguintes etapas: preparação, operação e validação dos dados coletados.

7.1.3.1 Preparação

Nessa etapa, os estudantes se comprometeram com o experimento e foram informados de sua finalidade. Assim, eles aceitaram os termos relacionados por meio da assinatura de um *Termo de Consentimento* com a confidencialidade dos dados que seriam coletados, i.e., seriam para fins acadêmicos e eles teriam liberdade para retirar-se (Apêndice A). Além desse formulário, outros objetos foram fornecidos como segue:

- *Formulário de Caracterização*: um questionário em que os sujeitos tiveram que avaliar o seu conhecimento sobre as tecnologias e os demais conceitos utilizados no experimento (Apêndice B).
- *Roteiro de atividades e formulário de Coleta de Dados:* o roteiro continha os passos para reestruturar os subdomínios *Rental* e *Trade* com AspectJ e Templates Acceleo. O formulário continha espaços vazios para serem preenchidos com os tempos de início e de término de cada atividade realizada durante do experimento, além dos problemas encontrados (Apêndice C).
- *Formulário de Opinião, Dificuldades e Sugestões:* documento contendo questionamentos para sabermos a visão dos alunos após a execução do experimento piloto (Apêndice D).
- *Formulário Final:* documento contendo questionamentos quanto à facilidade na utilização das duas tecnologias, em qual delas obtiveram mais erros e qual a sugestão para minimizar os erros obtidos. Esse formulário foi aplicado após o experimento real (Apêndice E).

Para reduzir a interferência do tempo consumido no aprendizado das tecnologias POA e MDD para modularizar os subdomínios de um FMD, um treinamento de dois dias com a duração de 4 horas por dia, foi planejado e conduzido. Assim, todos os sujeitos foram capazes de realizar as atividades propostas no experimento.

A plataforma adotada para realizar o experimento consistiu na utilização de Java como linguagem de implementação, AspectJ como a linguagem orientada a aspectos, templates Acceleo como ferramenta para criar as regras de geração de código fonte e a IDE Eclipse como ambiente de desenvolvimento.

7.1.3.2 Operação

Primeiramente, os sujeitos participaram de um treinamento, como descrito anteriormente. Conforme pode ser observado no Quadro 13, depois do treinamento, o experimento piloto foi executado. Durante o piloto, os sujeitos foram autorizados a fazer perguntas sobre qualquer dúvida quanto ao experimento. Os dados coletados nessa atividade serviram para rebalancear os grupos. Tanto no piloto, quanto no experimento real, os sujeitos tiveram que utilizar a documentação das classes, o diagrama de features (ilustrado na parte (c) da Figura 19), o mapeamento entre as classes e as features (Quadro 14), além dos comentários que foram inseridos nas classes que deveriam ser modularizadas. O experimento real foi realizado uma semana depois do experimento piloto. No experimento real, quando os sujeitos sinalizavam que tinham compreendido a tarefa, recebiam uma descrição adequada sobre o que tinha que fazer. O tempo começava a contar e os sujeitos não eram autorizados a fazer perguntas, pois isso poderia comprometer os dados coletados no experimento real.

7.1.3.3 Validação dos Dados

Os formulários preenchidos pelos sujeitos foram confirmados com os dados coletados no experimento piloto. Em geral, observou-se que os grupos desenvolveram as tarefas satisfatoriamente, e os dados obtidos estavam em conformidade com os limites esperados. Isso significa que os tratamentos foram executados corretamente e de acordo com o planejamento.

7.1.3.4 Dados coletados

O tempo despendido durante o processo de modularização de um FMD para uma LPF com ambas as técnicas, i.e., POA e MDD estão listadas na Tabela 1. O número de problemas também é mostrado nessa Tabela.

A Tabela 1 possui 10 colunas, “G” representa o grupo em que cada sujeito pertence e a coluna “S” lista o código de identificação de cada um deles. No grupo de colunas chamado “Tempo (min)” estão descritos os tempos em minutos utilizados pelos participantes para concluírem as modularizações com as técnicas MDD e POA. Os dados dessas colunas também são representados em um *ladder chart*, que é plotado na parte esquerda da Figura 32. Como pode ser visualizado na Figura 32, o tempo para obter uma LPF utilizando MDD tende ser menor do que o tempo despendido com POA. A coluna “Inconsistência” possui a quantidade de problemas obtidos com o uso das duas técnicas, por exemplo: aspectos introduzindo métodos e atributos em classes incorretas, regras incorretas de geração de código nos templates, classes estendendo classes inválidas que foram movidas para um novo pacote e com isso, torna-se necessário executar um “clean” para que os adendos dos aspectos entrecortem os *join points* corretamente. A coluna “Estrutura” especifica quando os desenvolvedores não desenvolviam um template ou um aspecto corretamente e assim, determinadas classes que dependiam de classes afetadas por esses templates/aspectos retornavam erros. Vale ressaltar que uma classe pode ser afetada por aspectos de maneira estática ou dinâmica, mas quanto aos templates, para descobrir sua corretude, deve-se gerar as classes a partir deles e isso é feito em tempo de projeto. Com as classes geradas é possível verificar se foram geradas adequadamente, conforme as regras descritas nos templates. A separação dos problemas nessas duas classificações serve apenas para indicar ao que se referem cada ocorrência.

Os valores dos problemas relacionados com “Inconsistência” e “Estrutura” são também representados no *ladder chart*. Os dados relacionados à inconsistência são representados na parte central e à estrutura, na parte direita da Figura 32, respectivamente. Note que o número de problemas representados neste *ladder chart* tende a ser maior com POA do que com MDD. Isso se aplica para “Inconsistência” e “Estrutura”. As próximas colunas *TotalMDD* e *TotalPOA*, referem-se ao total de problemas obtidos com templates e o total de problemas obtidos com POA,

respectivamente. Com esses dados torna-se claro que por meio de templates há uma redução do número de problemas encontrados nas LPFs resultante.

Tabela 1: Dados coletados

				Problemas				Número Total de Problemas	
		Tempo (min)		Inconsistência		Estrutura			
G	**S**	**MDD**	**POA**	**MDD**	**POA**	**MDD**	**POA**	**TotalMDD**	**TotalPOA**
1	**S1**	26	23	0	5	0	0	0	5
	S2	24	24	1	2	0	0	1	2
	S3	12	19	0	0	0	2	0	2
	S4	27	37	0	1	1	3	1	4
	S5	28	29	0	0	0	0	1	0
	S6	24	40	0	0	0	1	0	1
	S7	19	23	0	0	0	0	0	0
2	**S8**	25	13	1	0	2	0	2	0
	S9	12	15	0	1	0	1	0	2
	S10	21	19	1	1	0	2	1	3
	ºS11	41	47	0	4	0	0	0	4
	S12	26	32	0	0	0	2	0	2
	S13	30	35	1	1	0	1	1	2
	S14	31	34	0	1	1	0	1	1
	Avg.	23,46	26,38	0,31	0,92	0,31	0,92	0,61	1,84
		47%	53%	25%	75%	25%	75%	25%	75%

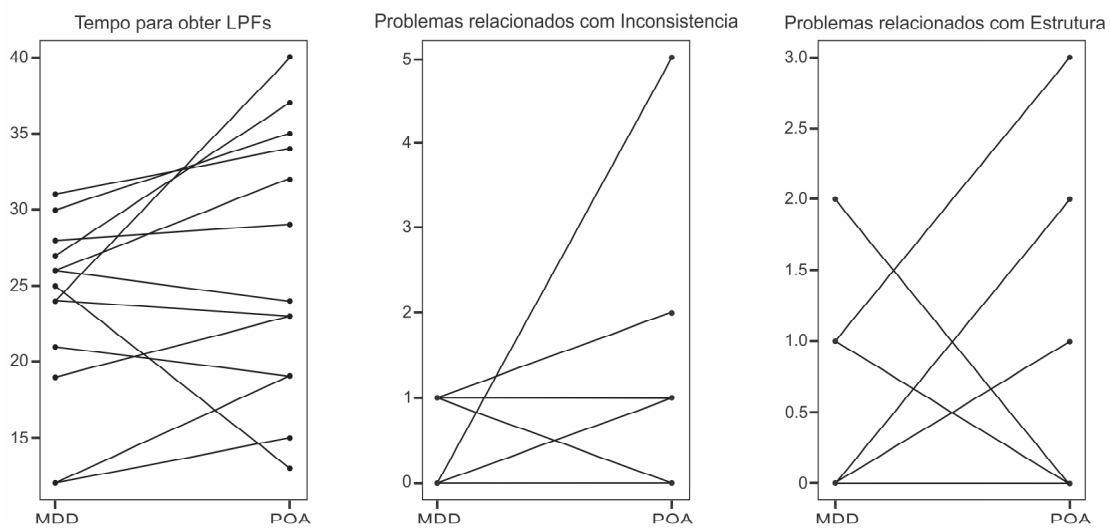


Figura 32: Tempo despendido e a quantidade de problemas encontrados durante a modularização dos subdomínios do GRENJ

7.1.4 Análise e Interpretação

Esta seção descreve os resultados experimentais. A análise é dividida em três pontos: (i) detalhamento estatístico, (ii) redução de dados e (iii) teste de hipóteses.

7.1.4.1 Detalhamento Estatístico

Os dados coletados durante o experimento são apresentados na Tabela 1, como: o tempo consumido por cada participante para modularizar o GRENJ em duas LPFs, com técnicas diferentes e o número de problemas encontrados nesse processo.

Na Tabela 1, pode-se observar que para a maioria dos sujeitos, a técnica POA levou relativamente 6% a mais de tempo para modularizar os subdomínios do GRENJ do que com templates, i.e., aproximadamente 53% contra 47%. Esse resultado deve-se ao fato de que com POA, os sujeitos obtiveram mais erros, 9 de 13 sujeitos considerados especificamente, concluíram a modularização com maior tempo. Por outro lado, quando os sujeitos utilizaram MDD, tiveram menos erros.

Além disso, na Tabela 1 é possível visualizar os dois tipos de problemas que foram encontrados no processo de modularização, relacionados à inconsistência e estrutura. Com os dados descritos na Tabela, fica evidente que por meio de MDD, todos os sujeitos obtiveram menos problemas do que com a técnica POA, i.e., 25% contra 75% para inconsistência e o mesmo para estrutura, respectivamente. Observando a Tabela 1, pode-se notar que MDD utilizando templates conduz o desenvolvedor a menos problemas do que a técnica de POA, considerando o escopo do experimento.

Considerando que os sujeitos utilizaram somente declarações intertipo para modularizar os subdomínios do FMD com POA, que são relativamente mais simples do que utilizar *join points* dentre outras abstrações, percebeu-se que muitos dos sujeitos, ao remover os métodos e atributos das classes para adicionar nos aspectos, se “perdiam” em não saber de quais classes tinham removido. Para evitar isso, conforme explicado no treinamento, os sujeitos deveriam primeiramente transferir os métodos e atributos que implementavam um interesse (subdomínio) em questão para um aspecto que foi fornecido. Nesse aspecto, as declarações intertipo tinham que ser implementadas considerando uma classe por vez. Com relação aos templates, os sujeitos não precisaram transferir trechos de código das classes para

outras unidades de implementação, pois bastava alterar o código fonte nos templates, inserindo rótulos nos fragmentos de código, associando-os ao seu subdomínio correspondente.

7.1.4.2 Redução de Dados

Antes de aplicar os métodos estatísticos, é necessário verificar a qualidade dos dados obtidos (Wohlin et al., 2000). A representação e a corretude dos dados possuem impacto direto nas conclusões que podem ser obtidas, caso dados incorretos sejam usados em uma análise estatística, provavelmente isso resultará em conclusões equivocadas. Dados incorretos podem ser provenientes de erros semânticos ou pela presença de *outliers*, que fornecem dados muito acima ou muito abaixo quando comparados com os dados restantes. Na Figura 33 ilustra-se um gráfico de caixa (*boxplot*) (Wohlin et al., 2000) que mostra a dispersão dos tempos despendidos por cada sujeito utilizando POA e MDD.

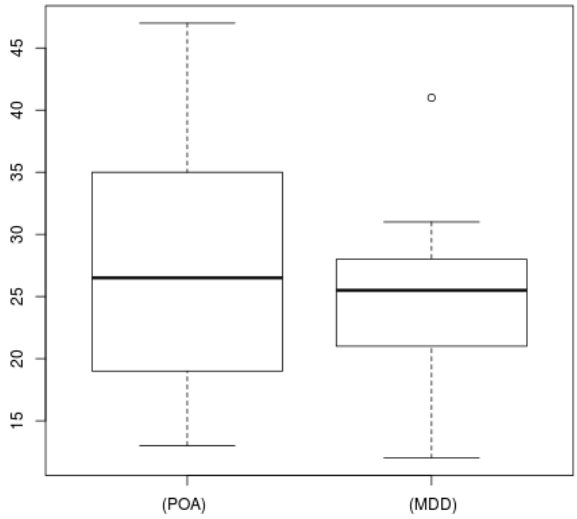


Figura 33: Box plot com o tempo despendido por cada sujeito no Experimento Real

No *bloxplot*, a linha sobre cada caixa marca a mediana do conjunto de dados. As hastes que se estendem acima e abaixo das caixas indicam o limite teórico dentro do qual provavelmente são encontrados todos os dados da amostra caso a distribuição for normal. Os valores que se encontram abaixo ou acima desses limites são representados como pontos individuais no gráfico, sendo caracterizados como *outliers.* Como ilustrado na Figura 33, tem-se a detecção de um *outlier,* o sujeito S11 (indicado pelo prefixo "o" na Tabela 1) que está representado por um pequeno

círculo. Assim, não o consideramos nas médias do tempo e da quantidade de problemas. Vale ressaltar que apesar do tempo despendido estar desbalanceado, os sujeitos são iniciantes quanto aos requisitos do experimento (Figura 31) e eles estão separados de forma balanceada, como descrito anteriormente.

7.1.4.3 Teste de hipóteses

O objetivo de testar as hipóteses consiste em verificar com um determinado grau de significância, se é possível rejeitar a hipótese nula (H_0) em favor de alguma hipótese alternativa.

Com relação ao tempo consumido pelos sujeitos durante o experimento, a QP1 implica em duas hipóteses (H_0 e H_1). Existem diversos tipos de testes estatísticos que podem ser aplicados em uma distribuição normal (Shapiro, S.S. e Wilk M.B., 1965). Antes de selecionar um teste estatístico, nós examinamos se os dados possuem uma determinada linearidade. Para isso, utilizamos o teste não paramétrico *Shapiro-Wilk* (Montgomery, 2000) que serve como parâmetro para avaliar a normalidade dos dados. Desse modo, considerando os tempos consumidos pelos 13 sujeitos (sem o sujeito S11) com as técnicas POA e MDD, descritos nas colunas agrupadas por “Tempo (min)”, obteve-se o valor 0,8107 para *p-value*. Como consequência não se rejeitou a hipótese de que os dados estão distribuídos de forma normal. A parte esquerda da Figura 34 ilustra um *Q-Q plot* (quantil – quantil plot) (Hazen, 1914) com a distribuição dos tempos consumidos pelos sujeitos.

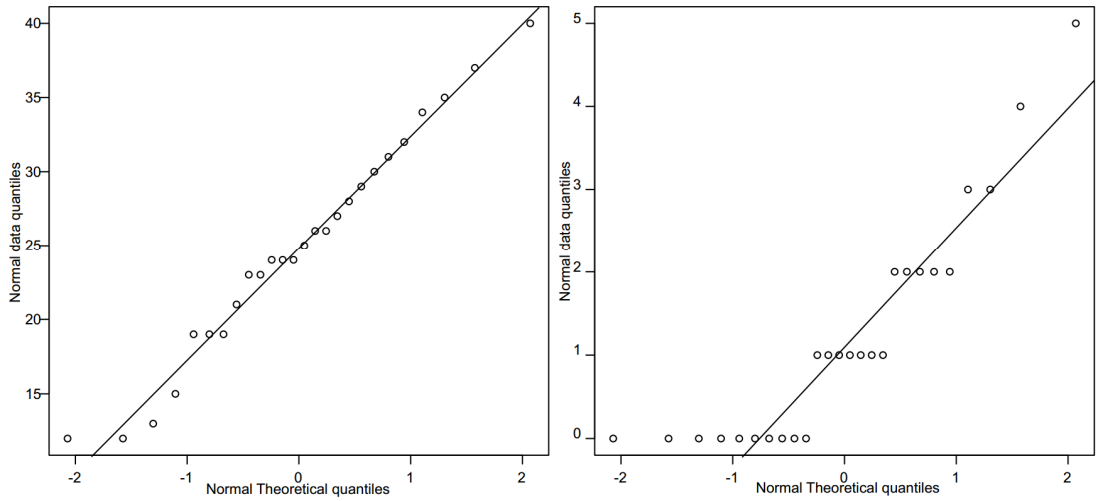


Figura 34: Teste de normalidade dos dados

O *Q-Q plot* é um método gráfico para comparação de duas distribuições de probabilidade por meio do cruzamento de seus *quantis* uns contra os outros. Para isso, o conjunto de intervalos para os quantis são escolhidos. Um ponto (*x, y*) no gráfico corresponde a um dos *quantis* da segunda distribuição (coordenada y) em função do mesmo *quantil* da primeira distribuição (coordenada x). Dessa maneira, forma-se uma linha parametrizada, cujo parâmetro corresponde ao intervalo para o quantil. Com a formação dessa reta inclinada, admite-se que as distribuições das amostras podem ser consideradas as mesmas.

Com a certificação de que os dados estavam distribuídos com normalidade, partiu-se para a aplicação de um *T-Test* pareado. Esse teste pode ser usado quando duas amostras de medidas repetidas precisam ser comparadas. Isso significa que essa avaliação pode ser feita com respeito a um sujeito mais de uma vez (Wohlin, 2000). Como em nosso caso, desejava-se comparar o tempo despendido por cada sujeito considerando duas técnicas para a modularização dos subdomínios de um FMD, considerou-se o emprego do *T-Test* pareado adequado.

Primeiramente foi necessário definir uma variável para diferença das amostras (d), da seguinte forma: d= { 2,1,-7,-10,-1,-16,-4,12,-3,2,-6,-5,-3}. Com isso, executou-se o *T-Test* e obteve-se *Sd* = 6,726336 e t_0 = -1,5669, com o intervalo de confiança entre -6.987761 e 1.141607. Após o cálculo de t_0, definiu-se o grau de liberdade (df), cujo valor corresponde a soma do número de dados nas amostras menos um (n-1), isto é, df=n-1=13-1=12. Além disso, considerou-se α = 0,05, ou seja, um risco de 5% em encontrar uma diferença significativa entre as médias das amostras. Logo, o nível de confiança do teste torna-se 95%. Diante disso, verificou-se a tabela padrão de distribuição de probabilidade estatística *t* de *Student* (Anexo A) (Gosset, 1947) para descobrir se a razão calculada é suficientemente grande de modo que se possa atestar que é improvável que a diferença amostral tenha sido mera casualidade. Assim, com base na tabela *t* de *Student*, $t_{0.025}$ = 2.179. Como $t_0 < t_{0.025}$ torna-se impossível rejeitar a hipótese nula com um teste bilateral com o nível 0,05, pode-se assumir estatisticamente que o tempo necessário para modularizar os subdomínios de um FMD em uma LPF usando as duas técnicas são aproximadamente equivalentes.

Com relação aos problemas, similarmente, utilizamos o teste não paramétrico *Shapiro-Wilk* com os dados da nona e décima coluna. Esses dados representam a quantidade de problemas encontrados pelos sujeitos durante a obtenção das LPFs,

utilizando as duas técnicas descritas anteriormente. Para esses dados, o *p-value* obtido foi de 0.001026, considerando α = 0,05. Como o valor de p-value é menor do que o nível alfa pode-se rejeitar a hipótese de que os dados são provenientes de uma população distribuída com normalidade. Observando a parte direita da Figura 34, torna-se evidente que a quantidade de problemas não está distribuída de forma padronizada. Assim, utilizamos um teste não paramétrico, *Wilcoxon signed-rank test*[14] (Wilcoxon, 1945), de modo que s/r = {-10, -2, -5.5, -8.5, -2, 8.5, -5.5, -5,5, -2}. Como resultado, obtemos o valor de *p-value* = 0.06549227 e como *p-value* possui um valor acima de 0.05, podemos concluir que não existe uma diferença considerável entre a quantidade de problemas entre os dois tratamentos. Assim, de acordo com o *Wilcoxon signed-rank test*, não podemos rejeitar H_0, apesar do total de erros obtidos com MDD ser menor (9 contra 4, sem *outlier*), comparando com os erros obtidos com POA.

A análise dos dados deste estudo foi realizada utilizando o sistema *R*[15] e o *RKWard*[16], um *front-end* para o R.

7.1.4.4 Avaliação Qualitativa

Apesar do escopo do estudo ter se restringido ao contexto universitário, a partir de um ponto de vista de validade é factível afirmar que as atividades de modularização de subdomínios por meio de templates, possuem uma ligeira tendência de colaborarem para maior produtividade, tanto com relação ao tempo despendido quanto a quantidade de erros, em comparação com POA.

Para validar essas observações junto aos sujeitos do estudo, após a execução do experimento os alunos receberam um formulário de avaliação final (Apêndice E), para relatarem sua percepção em relação ao uso das duas técnicas na modularização de features no contexto de LPFs. Com relação à dificuldade no emprego das técnicas, 64,28% dos alunos consideraram que a implementação foi realizada de forma mais produtiva por meio de templates, 14,28% consideraram que foram melhores com AspectJ e 21,42% informaram que não há uma diferença significativa em relação a facilidade no emprego das duas técnicas. Dentre os

[14] http://www.vassarstats.net/wilcoxon.html

[15] http://www.r-project.org/

[16] http://rkward.sourceforge.net/

alunos, 71,42% responderam que obtiveram mais erros com AspectJ, 21,42% afirmaram que por meio de templates a ocorrência de erros é maior e apenas 7,14% responderam não ter sentido dificuldade com ambas técnicas.

Em suma, alguns alunos informaram que a dificuldade estava na identificação do tipo de erro após a modularização. Com templates, os participantes tinham que introduzir rótulos associando os fragmentos de código com suas respectivas features e depois gerar o código por meio de um modelo para verificar se haviam erros, como classes que dependiam de métodos/atributos que não foram gerados em outras classes ou trechos de código que não deveriam ter sido gerados. Com relação aos aspectos, os alunos informaram que após a implementação de algumas declarações intertipo, muitas classes que dependiam dos métodos/atributos que tinham sido encapsulados pelos aspectos, não estavam "encontrando" tais elementos. Isso gerava dúvida, se eles haviam implementado de maneira incorreta ou seria necessário compilar para verificar se as introduções estáticas por meio das declarações entrariam em ação, solucionando o problema.

7.1.4.5 Ameaças à Validade

Nesta seção são apresentados itens que podem afetar os valores e a conclusão do experimento apresentado neste capítulo.

1) Validade Interna:
    - Nível de experiência dos participantes: a variedade do conhecimento prévio dos participantes poderia afetar os resultados obtidos. Para evitar essa ameaça, dividimos os sujeitos em dois blocos equilibrados, considerando o nível de experiência e em seguida os dados preliminares. Durante o treinamento, os indivíduos foram treinados em como utilizar POA e MDD para modularizar os subdomínios de um framework, a fim de obter LPFs.
    - Produtividade em avaliação: existe uma possibilidade de que isso pode influenciar os resultados do experimento, pois os sujeitos tendem a pensar que estão sendo avaliados por meio dos resultados do experimento. No sentido de evitar essa influência, explicamos aos alunos que ninguém estava sendo avaliado e a participação deles seria considerada anônima.

- Instalações utilizadas durante o estudo: computadores e instalações diferentes podem afetar os tempos registrados. No entanto, os sujeitos utilizaram a mesma configuração, modelo, sistema operacional e em números iguais. Os participantes não foram autorizados a mudar de máquina durante a mesma atividade.

2) Validação pela Construção:
    - Expectativa às hipóteses: os sujeitos conheciam os pesquisadores e sabiam que a utilização da técnica MDD com templates Acceleo poderia ser mais ágil do que POA, o que poderia influenciar os resultados do experimento, tornando-o menos imparcial. Para evitar isso, conduzimos o experimento de forma que os sujeitos mantivessem um ritmo constante em todas as atividades.

3) Validade Externa:
    - Interação entre configuração e tratamento: é possível que os exercícios realizados não tenham sido precisos, no sentido de representar situações do mundo real. Apenas duas LPFs foram desenvolvidas e tinham a mesma complexidade. Com relação a essa ameaça, não conseguimos amenizá-la.
4) Validade da Conclusão:
    - Confiabilidade das métricas: refere-se a precisão das métricas empregadas durante as atividades. Para suavizar essa ameaça, apenas foi considerado o tempo consumido pelos sujeitos, que foram captados em formulários.
    - Baixo poder estatístico: o número de participantes pode ser pequeno para gerar dados estatísticos confiáveis. No sentido de suavizar essa questão, foram aplicados dois testes para análise de dados: Teste-T para analisar estatisticamente o tempo consumido para obter as LPFs e o *Wilcoxon signed-rank test* para analisar estatisticamente o número de problemas encontrados no processo de obtenção das LPFs.

7.1.5 Considerações finais

Com os dados obtidos nesse experimento, observa-se que o esforço para modularizar os subdomínios de um FMD, empregando POA e templates, no sentido de obter LPFs, foi equivalente. Isso se deve ao fato de que a modularização sendo de subdomínios, refere-se a features de granularidade grossa, logo possuem um nível de abstração maior. Dessa maneira, na utilização de POA foram empregados somente os recursos das declarações intertipo, considerando a complexidade do framework utilizado. Caso a modularização fosse realizada com features de granularidade fina, o número de features seria superior, bem como as possibilidades de combinações. Assim, para realizar a modularização nesse outro contexto, o esforço seria superior, envolvendo muitos aspectos, captura complexa de argumentos, entrecortes de métodos e construtores e etc. Assim, a quantidade de erros provavelmente seria superior, uma vez que as features afetariam uma infinidade de classes.

Com relação à utilização de templates, para modularizar tanto as features de granularidade fina quanto as de granularidade grossa, o procedimento seria o mesmo, inserir marcadores no código fonte, associando os fragmentos de código com as features de um modelo correspondente. Dessa maneira, o esforço para modularizar as features de granularidade fina em comparação com as features de granularidade grossa seria maior apenas em virtude da quantidade de features, mas não em relação ao uso de artifícios para tratamentos mais complexos.

Capítulo 8

TRABALHOS RELACIONADOS

Neste capítulo são descritos alguns trabalhos que fornecem princípios da terminologia "Linha de Produtos de Frameworks" e outros que são relevantes no contexto de modularização de frameworks orientados a objetos para suavizar dificuldades como: (1) inflexibilidade em termos de composição e decomposição de features, (2) replicação de código desnecessário, além do (3) entrelaçamento e espalhamento de código que dificultam a evolução e a manutenção dos frameworks. Para evitar tais problemas, esses trabalhos apresentam técnicas, como: refatoração orientada a features e a refatoração por meio da orientação a aspectos.

8.1 Reestruturação de Frameworks

Batory et al. (2000) apresentam uma metodologia para decompor frameworks em colaborações, baseando-se na abordagem *Collaboration-based design*. Cada colaboração refere-se a uma feature cujas unidades possuem papéis definidos que colaboram com a implementação da feature. Com a composição dessas features, classes abstratas são refinadas e fornecidas para instanciação, possibilitando a redução de trechos de código replicados e desnecessários.

Com base na composição de um conjunto de features direcionado aos requisitos de uma determinada aplicação, pode-se suavizar o problema conhecido por "*overfeaturing*", isto é, a presença de variabilidades (features opcionais) desnecessárias do framework na aplicação final, uma vez que as aplicações resultantes da instanciação do framework conteriam código replicado que não necessariamente estaria sendo utilizado. Outro problema é a proliferação de

frameworks, que consiste na criação de várias versões do framework contendo variações nas classes, na tentativa de tornar a utilização do framework flexível, o que pode ocasionar problemas de manutenção.

Para realizar a decomposição do framework, deve-se inicialmente identificar as features primitivas. Apesar do trabalho não apresentar claramente como identificá-las a partir do código do framework, os pesquisadores mostram superficialmente essa identificação, baseando-se em uma hierarquia de classes. Por exemplo, se partindo de uma feature genérica (contendo classes abstratas), podem existir outras features contendo classes que refinam as classes da feature genérica, e assim por diante, a feature genérica pode ser admitida como primitiva.

Na Figura 35 demonstra-se hipoteticamente a decomposição de um framework em quatro features: L1, L2, L3 e L4. A feature L1 encapsula três classes, cada uma é raiz de uma hierarquia de subclasses. A feature L2 encapsula três classes, duas refinam classes existentes em L1 e a terceira (à direita) pertence a outra hierarquia. A feature L3 também encapsula três classes, sendo que duas delas refinam classes de L1 e L2. Finalmente, a feature L4 encapsula duas classes, ambas refinam classes existentes. A classe com a notação "*class Left*" na feature L4, colabora com a implementação de três features L1, L2 e L4, nas quais desempenha papéis diferentes. A feature L1 possui as classes abstratas (superclasses) nessa hierarquia, logo pode ser considerada como feature primitiva, enquanto que L2 e L3 possuem classes que refinam as classes de L1. Observa-se que a classe à direita na L4 refina uma classe da L3, iniciando uma nova hierarquia.

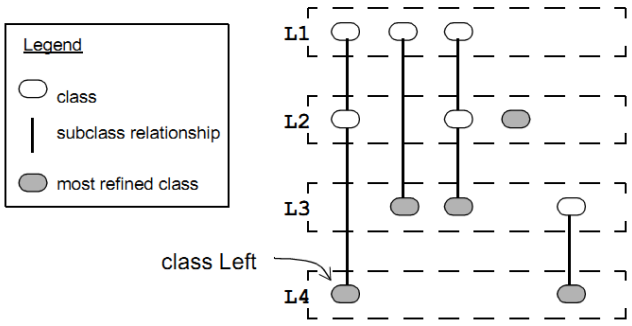


Figura 35: Hierarquia das features (Batory et al., 2000)

Dessa maneira, com o framework decomposto em features podem-se criar membros contendo um conjunto restrito de features. Esse membro para ser reutilizado ainda deverá ser instanciado em uma aplicação em desenvolvimento.

Sendo assim, ao invés da aplicação desenvolvida considerar o framework inteiro, a arquitetura final da aplicação, poderá conter um membro derivado do framework contendo um conjunto restrito das features. Por exemplo, supõe-se que uma determinada aplicação em desenvolvimento requeira a feature L3, o que implica em também utilizar as features L1 e L2, descartando a feature L4. Com a seleção da L3 e sua composição com as features que estão em nível superior, obtém-se apenas 4 classes que formam um membro do framework que pode ser instanciado pelas classes da aplicação base.

Na parte (a) da Figura 36 ilustram-se as hierarquias de herança representadas na Figura 35, agora considerando a instanciação do framework. Supõe-se a existência de uma linha entre L2 e L3, onde as classes acima da linha definem as classes do framework: A1, A2, A3 e A4. Observa-se que essas classes correspondem as mais refinadas, conforme as cadeias de refinamento que se encontram acima da linha. Abaixo da linha, as classes mais refinadas são classes concretas de instância do framework: C1, C2, C3 e C5. Vale ressaltar que A4 não precisa ser necessariamente refinada. A parte (b) da Figura 36 representa o resultado de uma composição, na qual somente as classes mais refinadas permanecem.

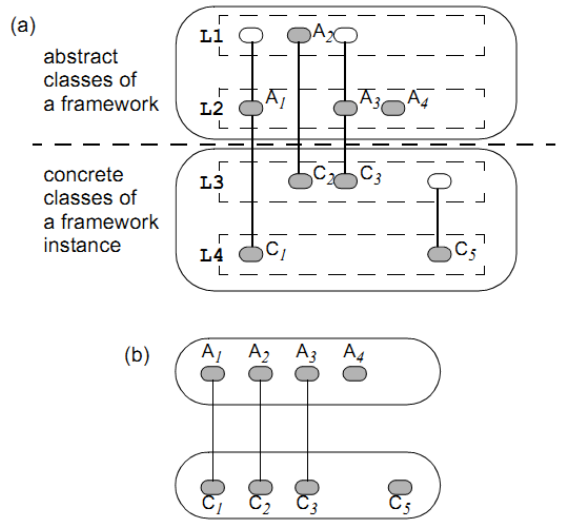


Figura 36: Refinamento de hierarquias e instâncias de framework (Batory et al., 2000)

Com a possibilidade de compor membros conforme as necessidades de uma determinada aplicação podem-se evitar trechos de código desnecessários nas classes do framework, a presença de variabilidades que não serão utilizadas pela

aplicação, bem como a replicação de código desnecessário, tendo em vista a utilização do framework por diversas aplicações.

Dessa forma, conclui-se que por meio da decomposição de frameworks em features, podem-se criar instâncias de framework somente com as classes refinadas e prontas para utilização por uma aplicação em desenvolvimento e, além disso, múltiplas classes dessas instâncias podem ser atualizadas com a inclusão ou remoção dessas features sem precisar modificar manualmente o código. Esse tipo de modularização torna a utilização de frameworks flexível em termos de composição, pois as features podem ser acopladas ao código da aplicação de forma gradual, conforme necessário.

Kulesza et al. (2005) apresentam um processo genérico para reestruturação de frameworks utilizando orientação a aspectos, visando obter melhor separação de interesses transversais e redução do entrelaçamento e espalhamento de código. Como estudo de caso, foi utilizado o framework JUnit[17] que apesar de ser relativamente pequeno, é amplamente conhecido por permitir o projeto, implementação e execução de testes de unidade em aplicações implementadas com a linguagem Java.

Com relação a refatoração do framework, seguiu-se os seguintes passos: (1) compreensão da estrutura e comportamento do framework, (2) análise de principais componentes e identificação de interesses que entrecortam o código fonte, (3) refatoração do framework com aspectos, separando as features transversais e (4) análise comparativa entre o framework modificado e a versão anterior orientada a objetos.

No primeiro passo, constatou-se a existência de três componentes principais: *testing*, *runner* e *extensions*. O componente *testing* contém classes responsáveis por especificar o comportamento básico para execução de casos e suítes de testes. Os principais pontos de extensão (*hot-spots*) disponíveis neste componente são implementados pelas classes *TestCase* e *TestSuite*. Os usuários do framework precisam estender essas classes para criação de casos de testes específicos para suas aplicações. O componente *runner* é responsável por oferecer uma interface que facilita a execução e visualização do resultado de casos e suítes de testes. O

[17] http://junit.org/

componente *extensions,* por sua vez, é responsável por definir um conjunto de funcionalidades que estendem comportamentos básicos do framework JUnit.

No segundo passo, com a análise dos componentes, foi possível observar que o framework utiliza os padrões *Observer* e *Decorator*, e que havia entrelaçamento de código entre os módulos do framework, o que dificulta de certa forma seu entendimento, ou seja, o código referente ao padrão *Observer* encontrava-se entrelaçado com o código das classes *TestCase* e *TestResult,* dificultando a compreensão do comportamento do núcleo do componente *testing,* bem como o código de composição dos componentes *testing* e *runner*. Além disso, muitas classes do componente *extensions* são implementadas utilizando mecanismos de herança, dificultando a compreensão de como as classes de extensão (subclasses) são relacionadas com as classes internas do componente *testing.* Isso restringe os tipos de extensões que o usuário do framework pode definir para o componente *testing* no processo de instanciação do framework.

No terceiro passo, após a identificação das funcionalidades (*features*) entrelaçadas, foram analisadas diferentes alternativas de refatoração orientada a aspectos para identificar a mais adequada. A composição entre os componentes de *testing* e *runner* utilizava o padrão *Observer* e com a refatoração OA foram substituídos por “aspectos observadores”, que são aspectos responsáveis por notificar o status de execuções dos casos e suítes de teste. O componente de *extensions* foi implementado com um conjunto de aspectos que são utilizados para especificar as classes do framework (como as classes, *TestCase* e *TestSuite*). Esses aspectos introduzem o comportamento transversal relacionado com a execução dos casos de teste.

No quarto passo, por meio da análise comparativa entre a versão inicial do framework com a versão modularizada com aspectos, foram observadas duas melhorias principais: (1) melhor separação dos componentes (*features*) que constituem o framework e (2) maior flexibilidade para especificar a composição com estes componentes. Com o framework reestruturado, tornou-se possível definir novos *pointcuts* para componentes de teste que podem ser utilizados para notificar elementos específicos durante a execução. Além disso, os aspectos estendem o componente de teste, sendo assim, os usuários do framework não precisam definir subclasses específicas, apenas informam por meio de aspectos, quais funcionalidades devem ser testadas.

O estudo explanou que a refatoração de frameworks utilizando orientação a aspectos possibilita realizar a reestruturação, evitando o entrelaçamento de código e criando módulos com uma concepção mais clara. Conforme a pesquisa realizada, obteve-se níveis mais efetivos quanto à modularização dos componentes do framework e flexibilidade para especificar novos alvos para teste. Dessa forma, nota-se que esse tipo de reestruturação de frameworks favorece a melhor utilização das funcionalidades disponíveis, em termos de facilidade de utilização. No nosso caso, poderíamos averiguar que se ao utilizar esse framework em um determinado conjunto de aplicações, geralmente todos esses componentes são utilizados juntos, então não separaríamos como três features distintas.

8.2 Linha de Produtos de Framework

Oliveira et al. (2012) apresentam o conceito de Linha de Produtos de Frameworks que permite a composição de um conjunto restrito de *features* conforme as necessidades da aplicação em desenvolvimento, com o objetivo de tornar a utilização de frameworks mais flexível, quando comparado à utilização convencional de frameworks, em que as aplicações geradas consideram a arquitetura integral do framework, comprometendo a legibilidade do código e dificultando a manutenção e evolução dessas aplicações. Vale ressaltar que este trabalho é anterior ao desenvolvido neste mestrado.

Similar ao conceito de LPS, como discutido anteriormente, os frameworks promovem o reúso de software oferecendo variabilidades que são genéricas e servem para o desenvolvimento de um determinado conjunto de aplicações. Em geral, os frameworks mostram-se inflexíveis quanto à composição e a decomposição dessas variabilidades, pois as aplicações derivadas com sua utilização geralmente incluem a arquitetura completa do framework, independentemente das necessidades específicas das aplicações, afetando a manutenção e a evolução.

A fim de lidar com essa dificuldade, os pesquisadores apresentam as características de uma LPF que visa à redução do número de *features* que o engenheiro de software deve selecionar enquanto cria uma determinada configuração (que ainda consiste em um framework), evitando código desnecessário

nas aplicações finais. Além disso, apresenta os passos utilizados para a obtenção de uma LPF a partir de um framework existente. O processo proposto nessa dissertação é uma evolução desses passos.

Na Figura 37 representa-se a visão geral de uma LPF em que o círculo no topo representa uma LPF dividida em várias partes. As *features* obrigatórias estão agrupadas com o rótulo “A” e os grupos rotulados do “B” ao “P” representam *features* opcionais e alternativas. Com a combinação das *features* alternativas e opcionais podem ser geradas diferentes configurações. Os retângulos “F1”, ”F2” e “F3” representam três possibilidades de configurações. Cada uma dessas combinações contém obrigatoriamente as *features* “A” e outras *features* que fazem parte da LPF. As configurações requerem que o engenheiro de software crie módulos adicionais (como representado por “APP3” e “APP4”). Por exemplo, as aplicações derivadas da configuração “F2” não utilizam todas as *features*, somente as necessárias. As regras de composição podem ser utilizadas para validar configurações da LPF e determinar o número possível de configurações de frameworks.

A visão *top-down* aborda um domínio geral de aplicações, cobrindo várias funcionalidades, sendo mais adequada para organizações de software que pretendem distribuir seus frameworks, enquanto que a abordagem *bottom-up* considera a necessidade de uma seleção restrita de *features* para aplicações. No primeiro nível de abstração, pode-se visualizar uma LPF. O segundo nível inclui algumas estruturas derivadas da LPF, enquanto no terceiro nível de abstração pode-se instanciar essas estruturas para criar aplicações com um conjunto restrito de *features*.

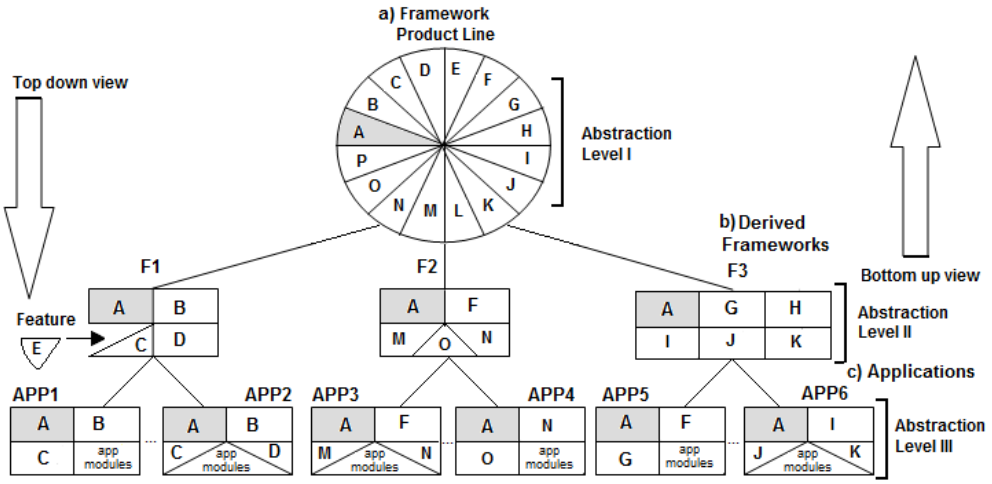


Figura 37: Visão geral de uma LPF (Oliveira et al., 2012)

Com base na abordagem apontada, os pesquisadores conduziram um estudo de caso com o intuito de reestruturar o framework de aplicações GRENJ, em uma LPF. Conforme comentado anteriormente, esse framework é baseado em uma linguagem de padrões denominada GRN que trata do domínio de gerenciamento de recursos de negócios. Vale ressaltar que o processo de reestruturação utilizado foi orientado por meio dessa linguagem de padrões. No processo *FMDtoLPF* proposto nesta dissertação, o mesmo framework foi modularizado considerando além da documentação, seu histórico de utilização por um conjunto de aplicações.

A estratégia utilizada nesse estudo de caso consistiu em agrupar os módulos do framework que possuíam um mesmo interesse de implementação em *features*. Porém, foi observado que não havia um conjunto de classes (pacotes, por exemplo) lidando com apenas uma característica específica de implementação, pelo contrário, os interesses estavam espalhados e entrelaçados entres os módulos do framework. Para enfatizar esses problemas, em cada *feature* do GRENJ mostrada na Figura 38, existem rótulos que indicam a presença de código relacionado com outras *features*, indicando o espalhamento e entrelaçamento. Por exemplo, a implementação da *feature* "*Resource*" (R) está espalhada pelas *features* "*Resource Quantification*" (RQ), "*Single Resource*" (SR), "*Measurable Resource*" (MR) e "*Instantiable Resource*" (IR).

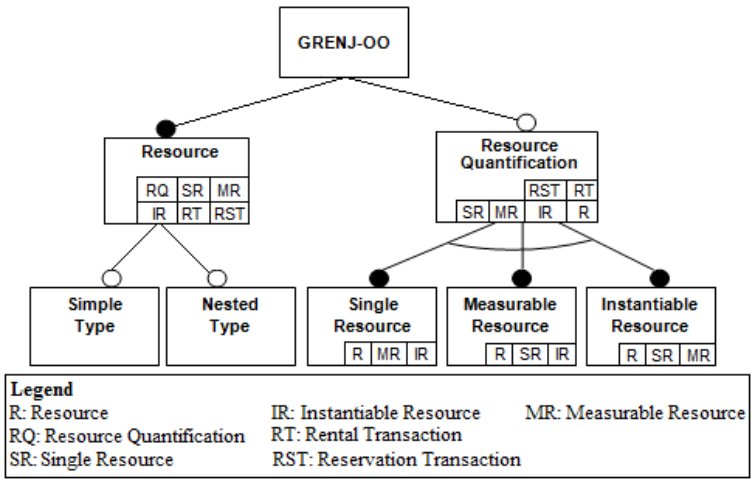


Figura 38: Modelo de features do GRENJ com os interesses espalhados e entrelaçados (Oliveira et al., 2012)

Observando os interesses com esses problemas, foi aplicado o conceito de LPF para modularizar o GRENJ utilizando a POA, removendo os trechos de código que tornavam os interesses espalhados e entrelaçados, e encapsulando-os em

aspectos. O framework possui um amplo número de variabilidades e com a utilização da abordagem, tornou-se possível derivar sub configurações do framework utilizando composição de *features*. O engenheiro de software seleciona as *features* necessárias, criando uma configuração válida e a partir desse subconjunto de *features*, acopla aos módulos específicos das aplicações, obtendo sistemas com somente as *features* necessárias.

Mesmo consistindo em resultados preliminares, o estudo apresentou-se viável por suavizar o problema de código desnecessário em aplicações provenientes da utilização de frameworks, melhorando a estrutura tanto do framework como das aplicações geradas em termos de manutenção e evolutibilidade. Entretanto, percebeu-se que os princípios das LPFs poderiam ser mais explorados. Desse modo, este projeto de mestrado contém refinamentos dos conceitos de LPFs, evidências das melhorias obtidas por meio de LPFs e detalhamento de um processo de modularização de FMD para LPFs, considerando cenários de utilização que as LPFs devem atender no final do processo.

Batory e Shepherd (2011) apresentam uma abordagem para a obtenção de uma Linha de Produtos de Linha de Produtos, a partir de uma Linha de Produtos existente. A finalidade dessa abordagem consiste em fornecer especificações mais simples, melhorar o gerenciamento da complexidade e evitar redundâncias em LPS, cuja parte comum de todas as aplicações derivadas, isto é, o *core* é abrangente e possui features que poderiam ser opcionais. Esse fato implica na construção de aplicações que consideram funcionalidades presentes no core que não necessariamente estão sendo utilizadas e, além disso, impede que uma aplicação que possua um domínio mais restrito considere apenas parte do *core*.

Segundo os pesquisadores, trata-se de uma ocorrência comum em LPS e para lidar com isso, propõe-se uma abordagem de reestruturação que consiste na construção de novas *features* que mapeiam as interações entre as *features* que são “sobrepostas”, isto é, a sobreposição de diferentes particionamentos de uma mesma base de código, aumentando o número de variabilidades possíveis e formando uma Linha de Produtos de Linha de Produtos, ou seja, uma LPS de rank 2.

Considera-se que uma LPS é de *rank* 1 (LPS^1) quando a composição de suas *features* geram aplicações, enquanto que a composição de *features* em uma LPS de rank 2 (LPS^2) gera outra LPS com um subconjunto restrito de *features* e não aplicações, isto é, uma LPS^1 que é uma linha de produtos menor a partir de LPS^2.

Para apresentar a abordagem proposta, foram realizados alguns experimentos. Inicialmente adotou-se uma linha de produtos denominada Linha de Produtos de Operações (*Operations Product Line* - OPL), que implementa expressões aritméticas que podem ser impressas e avaliadas. Por exemplo, a impressão de (add 4 5) retorna uma string “4 + 5” enquanto a avaliação retorna 9. O código da OPL é mostrado na Figura 39. A OPL possui três *features*: *core, print* e *eval*. A posição em que um fragmento de código é inserido denomina-se ponto de variação (PV), designado com n♦. Todos os PVs são pareados com seus respectivos fragmentos, rotulados por @n. Observa-se que nessa linha de produtos há *features* sobrepostas, por exemplo, na classe *Plus* e *Times* há trechos da *feature eval*, *print* e *core.*

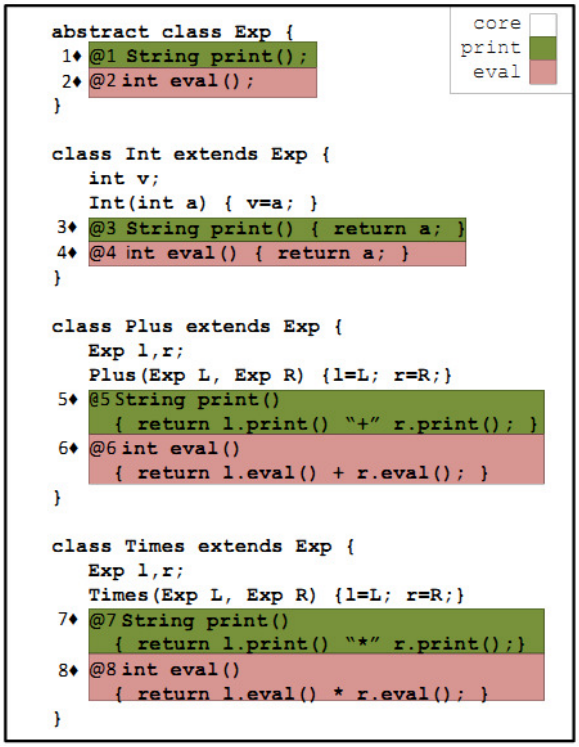


Figura 39: Código da OPL com pontos de variação (Batory e Shepherd, 2011)

Na Figura 40 ilustra-se o conteúdo separado de cada *feature*. A OPL é uma LPS[1], cujos módulos consistem em fragmentos de código. O módulo *core* contém uma hierarquia enraizada pela classe *Exp* com subclasses concretas: *Int*, *Plus* e *Times*. Nenhuma classe em *core* tem métodos. O módulo *print* adiciona o método print() para cada classe assim como o módulo *eval* adiciona o método eval().

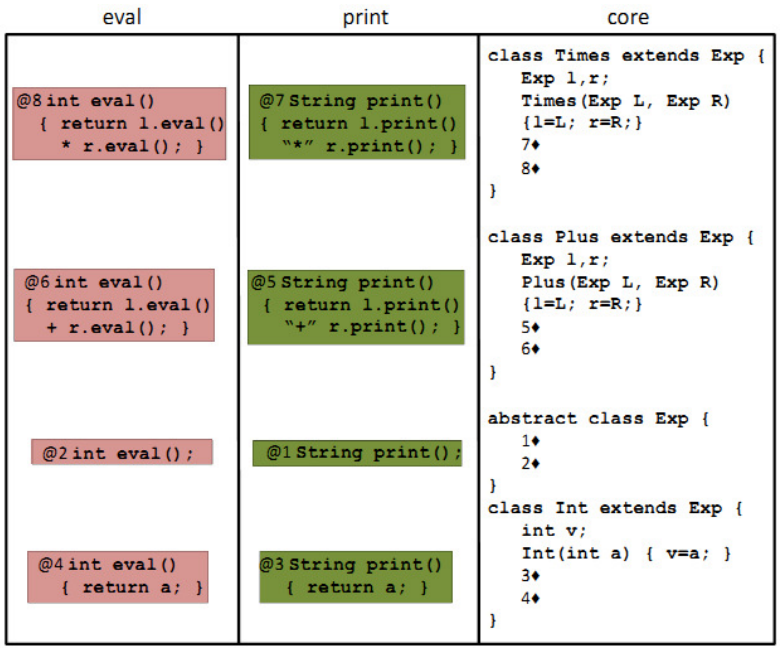


Figura 40: Conteúdo das features da OPL (Batory e Shepherd, 2011)

Na Figura 41 ilustram-se as três aplicações válidas, geradas a partir da composição (⊕) de *features* da OPL, dentre as quatro sentenças possíveis: *"core", "print⊕core", "eval⊕core" e "eval⊕print⊕core"*. Vale ressaltar que há uma restrição quanto à composição de *features* na OPL: a aplicação gerada que contiver o módulo *eval* deve obrigatoriamente incluir o módulo *print*. O programa *core* não pode imprimir nem avaliar expressões, enquanto que, um programa *"print⊕core"* pode imprimir expressões e um programa *"eval⊕print⊕core"* pode imprimir e avaliar expressões.

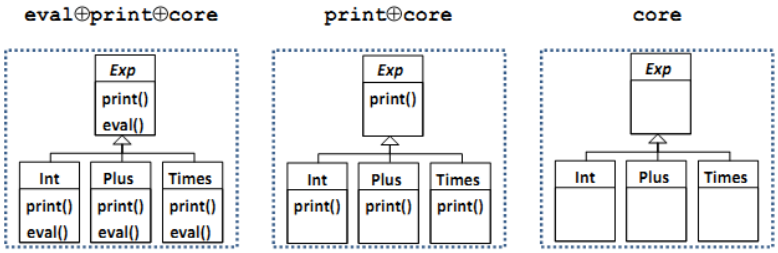


Figura 41: Composição de features na OPL (Batory e Shepherd, 2011)

Observa-se que as funcionalidades de *Plus* e *Times* são obrigatórias e pertencem ao *core* de OPL, portanto, fazem parte de todas as aplicações derivadas de OPL. Para reestruturar a OPL em uma LPS^2, possibilitando gerar aplicações que possuam apenas *Plus* ou *Times*, é necessário tornar *Plus* e *Times* funcionalidades opcionais. Para isso, são necessários dois passos principais. O primeiro passo

consiste em observar todas as aplicações válidas da OPL, conforme mostrado na linha superior da Figura 42. Em seguida, o segundo passo consiste em remover o código referente às classes que devem se tornar opcionais, no caso da OPL, remove-se o código referente à classe *Times* conforme mostrado na linha do meio da Figura 42 e depois, remove-se o código da classe *Plus*, formando seis novos programas, resultando em nove programas possíveis que constituem uma LPS^2, a Linha de Produtos de Problemas de Expressão (EPL).

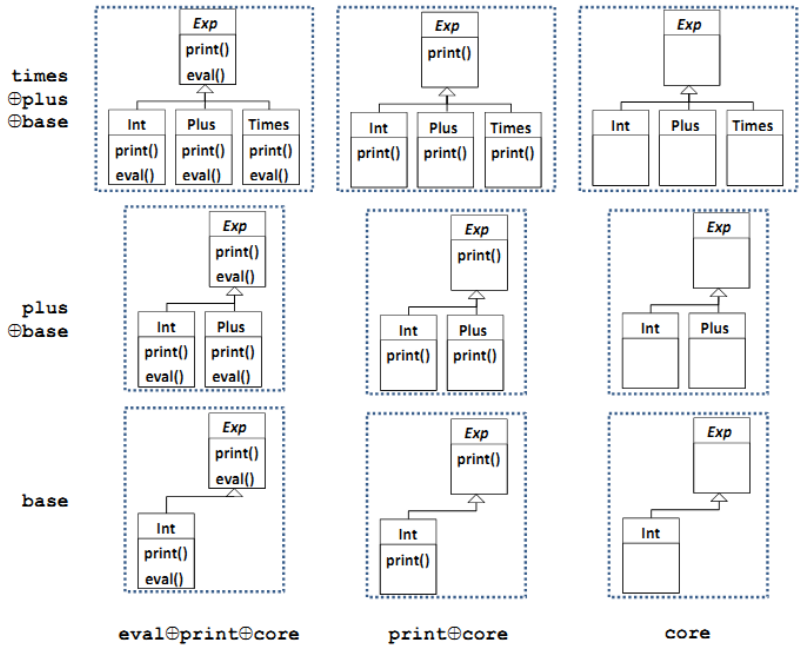


Figura 42: Produtos da EPL (Batory e Shepherd, 2011)

A ideia dessa reestruturação consiste em decompor o *core* da OPL observando o que pode se tornar opcional, para obter um *core* mais compacto, gerando novas *features,* resultando na obtenção de mais variações possíveis. Observando a Figura 42, a EPL possui dois eixos de variabilidade: (a) primeiro eixo formado pela coluna de *features*, especificadas por *print* ou *eval* e (b) outro eixo definido pelas linhas de *features*, especificadas pelas funcionalidades *Plus* ou *Times* presentes em programas da OPL. Pode-se afirmar que EPL é resultante da concatenção desses dois eixos de variabilidades. O primeiro é formado pelo modelo de *features* [eval] [print] core e o segundo formado pelo modelo de *features* [times] [plus] base.

Segundo Batory e Shepherd (2011), um modelo de *features* de uma LPS de rank 2 nem sempre pode ser a concatenação de dois modelos de *features* de uma

LPS de rank 1. Pode ser que haja restrições que impeçam ou exijam certas combinações que não façam parte dos modelos de rank 1. No caso da EPL, não há essa complexidade.

A variabilidade extra de EPL em relação a OPL é obtida pela adição de PVs no código da OPL. Na Figura 43 ilustra-se a atualização do código base. O PV +♦ é par do fragmento do código que contém a classe *Plus*, com 5♦ e 6♦ e o PV *♦ é par do fragmento do código que contém a classe *Times*, 7♦ e 8♦. Observa-se que *Plus* e *Times* tornaram-se novas *features* e estas são opcionais e mapeiam a sobreposição de *features* existentes.

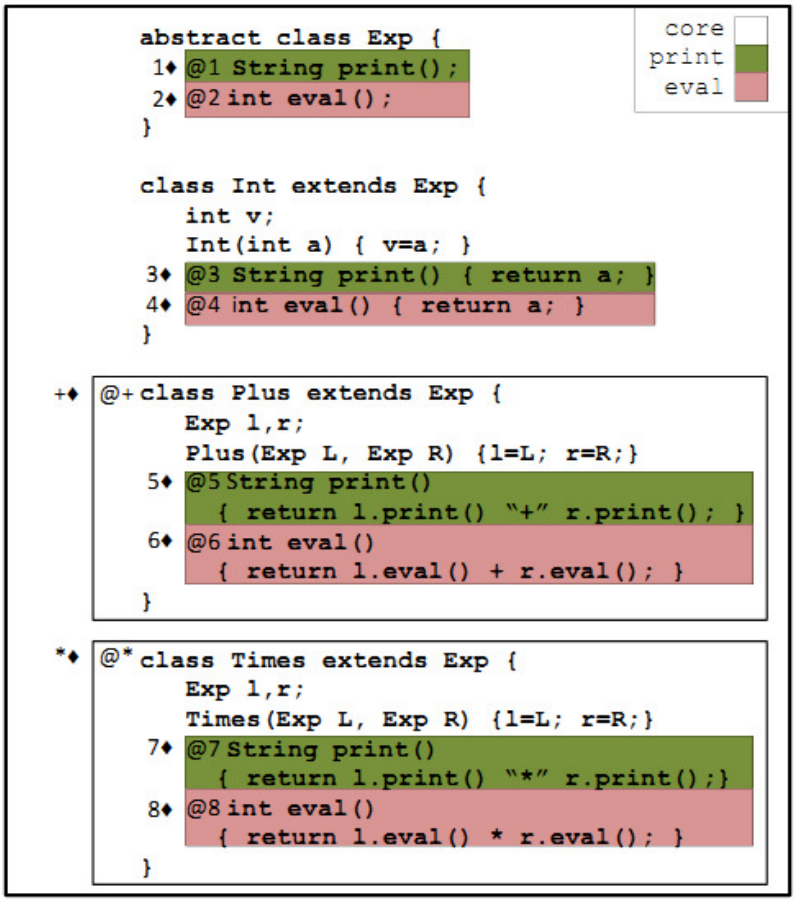


Figura 43: Código da EPL com pontos de variação (Batory e Shepherd, 2011)

Nota-se que a EPL constitui uma LPS de rank 2, visto que permite compor uma linha de produtos menor e a partir dessa segunda linha de produtos pode-se gerar aplicações. Por exemplo, criando um membro da EPL contendo as *features print* e *eval* pode-se gerar aplicações contendo somente a *feature* Plus, contendo *Plus* e *Times* ou uma aplicação sem as *features Plus* e *Times*. O precursor da EPL (a OPL) não possibilita a geração de tais aplicações.

Os módulos de uma LPS[2] podem ser agrupados em uma matriz 2D e as expressões com estes módulos podem ser criadas eliminando as linhas e colunas desnecessárias. A Figura 44 mostra o mapeamento dos fragmentos dos códigos mostrados na Figura 43 com uma matriz. Por exemplo, o elemento (*plus*, *print*) implementa as funcionalidades *Plus* e *print*, ou seja o método print() na classe *Plus*. A notação "≠" refere-se à interação entre as *features*.

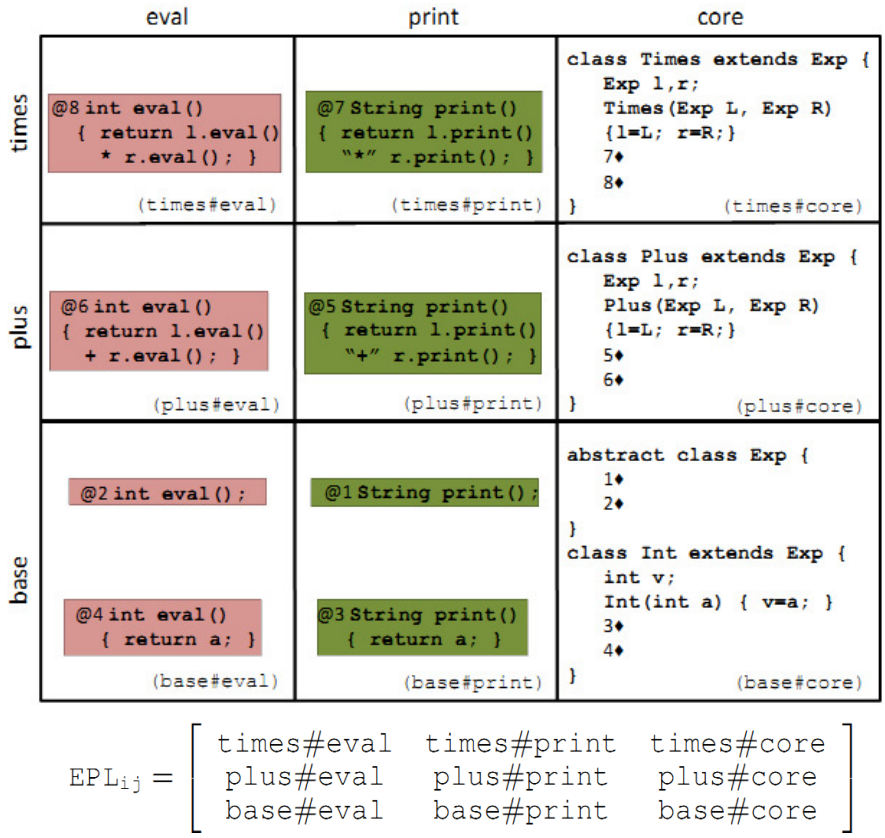


Figura 44: Mapeamento dos módulos das features da EPL com elementos da matriz (Batory e Shepherd, 2011)

Conforme observado, o processo de reestruturação da OPL em uma linha de produtos de linha de produtos consistiu em observar uma sobreposição de dois diferentes particionamentos que pertenciam a uma única base de código, ou seja, a mesma linha de produtos. Um particionamento composto por *print*, *eval* e *core* e o outro particionamento composto por *times*, *plus* e *base.* Após a criação de todos os membros possíveis de OPL, notou-se que *Plus* e *Times* poderiam ser removidos do *core* para se tornarem *features* opcionais.

Com isso, obtiveram-se mais variabilidades, ampliação da quantidade de membros possíveis e a possibilidade de gerar membros com somente as *features*

necessárias. Assim como uma LPS possuindo um *core* extenso pode ser reestruturada para fornecer especificações mais simples e evitar redundâncias, similarmente no contexto de frameworks, torna-se coerente “quebrar” seu *core* em outras *features*, e para isso, podem ser criados todos os membros possíveis do framework, isto é, as combinações válidas com as *features* disponibilizadas pelo framework, e assim, analisar quais *features* que pertencem ao *core* podem ser opcionais, possibilitando novas variabilidades e membros mais restritos contendo somente as *features* indispensáveis para o acoplamento com o código base da aplicação, obtendo dessa forma, melhorias quanto à modularidade, manutenção e reúso.

8.3 Considerações Finais

No trabalho de Batory et al. (2000) apresenta-se alguns princípios da terminologia "Linha de Produtos de Framework", isto é, a possibilidade de criar instancias de frameworks, com uma parcela das features disponibilizadas. Para isso, o framework deve ser decomposto em features, as quais sendo compostas podem resultar em frameworks menores e direcionados aos requisitos das aplicações, evitando variabilidades desnecessárias.

Kulesza et al. (2005) apontam que a modularização de frameworks pode ser feita por meio de orientação a aspectos, caso o objetivo seja modularizar interesses transversais e reduzir o entrelaçamento/espalhamento de código. Além dessas melhorias, pode-se melhor o reúso do framework, por meio dos mecanismos abstratos de composição fornecidos pela POA. No nosso caso, a ideia consiste em modularizar frameworks melhorando sua flexibilidade arquitetural, de modo que o EF seja capaz de selecionar um conjunto menor de features e obtenha frameworks menores e voltados a um subdomínio e/ou requisitos de uma determinada aplicação, sem a presença de features que podem nunca ser utilizadas.

Batory e Shepherd (2011) apontam a necessidade de decompor o *core* das LPS, uma vez que após uma série de evoluções podem existir variabilidades no core que deveriam ser opcionais. Com essa decomposição, novos produtos podem ser criados contendo somente a parte necessária. Com relação ao reúso de frameworks,

podem ser considerados como um "*core* indivisível", uma vez que grande parte das variabilidades opcionais, mesmo que nunca sejam utilizadas, persistem em todas as aplicações desenvolvidas. Diante disso, torna-se coerente a aplicação dessa ideia de decomposição para frameworks, no sentido de torná-los flexíveis arquiteturalmente, permitindo que instâncias (membros) sejam construídas com uma parcela das variabilidades disponibilizadas.

No trabalho de Oliveira et al. (2012) apresentam-se os conceitos das LPFs e um estudo de caso no qual o framework GRENJ foi modularizado com o auxilio da linguagem de padrões GRN. A técnica empregada foi POA e o critério utilizado na modularização foi decompor todas as variabilidades possíveis. Como resultado obteve-se uma LPF e apesar das melhorias com relação ao entrelaçamento e espalhamento de código, a quantidade de aspectos foi exponencial. Os aspectos possuíam muitas replicações de código para tratar as features de granularidade fina, incluindo suas combinações e interferências. Dessa forma, apesar das melhorias que podem ser obtidas com POA e o conceito de LPF apresentado, apresentou-se nesta dissertação, refinamentos a esse conceito e o MDD (templates) como uma técnica alternativa para tratar as dificuldades em relação às features de granularidade fina. Além disso, foi proposto um processo de decomposição dessas features, fundamentação dos principais Cenários de Utilização que devem ser considerados nesse processo e evidências das melhorias que podem ser obtidas, baseando-se em estudos avaliativos.

Capítulo 9

CONCLUSÃO

O Reúso de software é uma estratégia de Engenharia de Software voltada para a reutilização de software existente. Embora o reúso tenha sido proposto como uma estratégia de desenvolvimento há mais de 40 anos, apenas a partir do ano 2000, o “desenvolvimento com reúso” foi considerado indispensável para o desenvolvimento de software. A mudança para o desenvolvimento com o reúso tem sido uma resposta às demandas por software com menor custo de produção e manutenção, rapidez na entrega de sistemas e aumento na qualidade de software. Cada vez mais empresas consideram seu software como um bem valioso e que promover sua reutilização, amplia o retorno de seus investimentos (Sommerville et al., 2011).

Nesse contexto, foram desenvolvidas diversas técnicas que apoiam o reuso de software com diferentes propósitos, tais como: padrões de projeto, LPS, desenvolvimento baseado em componentes, frameworks e etc. Dentre essas técnicas, os frameworks são amplamente empregados no desenvolvimento de software, tanto em partes de uma aplicação, como é o caso de frameworks de infraestrutura, quanto para desenvolver uma aplicação completa, com o apoio dos chamados frameworks de aplicação. Por exemplo, o framework Hibernate é utilizado para lidar com a persistência de dados em bancos de dados relacionais, enquanto que o framework JUnit apoia a realização de testes unitários em aplicações Java. Além de frameworks de aplicação como o GRENJ que permite desenvolver aplicações para gestão de recursos.

De modo geral, uma tendência comum dos frameworks em suas evoluções, consiste em melhorar a modularidade para facilitar a manutenção e agregar novas

features visando atender outros domínios de aplicações, ampliando sua utilização pelos usuários. Muitas vezes, quando não há um gerenciamento adequado nesse processo de associação de novas features, pode-se resultar em entrelaçamento e espalhamento de código, fazendo com que determinadas features extrapolem os limites dos domínios aos quais foram inicialmente projetadas, gerando o que chamamos de Frameworks de Múltiplos Domínios. Esses frameworks podem oferecer uma arquitetura inflexível quanto ao grau de composição e decomposição dessas features. Como consequência, o Engenheiro de Framework torna-se incapaz de atender demandas específicas, i.e., frameworks que sejam direcionados às necessidades de determinados conjuntos de aplicações. Assim, todas as aplicações desenvolvidas com o apoio desse framework, como o consideram integralmente, acabam incluindo todas as features dele, independentemente se houver features com mínimas chances de serem utilizadas futuramente. Afinal, é provável que grande parte dessas features dificilmente sejam utilizadas por determinados grupos de aplicações, uma vez que essas features são específicas de domínios isolados e diferentes do domínio desse grupo.

Diante disso, neste trabalho foi desenvolvida uma abordagem para modularização de Frameworks de Múltiplos Domínios em LPF, solucionando tais dificuldades. Com base em refinamentos dos conceitos de LPF e em suas possíveis formas de modularização, essa abordagem define diretrizes que auxiliam a modularização desses frameworks em LPFs. Dentre essas diretrizes, um dos pontos discutidos é quanto à granularidade das features, pois ao considerar o cenário de utilização que uma LPF deve satisfazer, implica-se em definir também como as features devem ser projetadas e organizadas. Dessa maneira, pode-se conduzir a modularização garantindo que os frameworks provenientes dessa linha estejam em conformidade com as possíveis demandas. Essas demandas correspondem a frameworks personalizados, contendo um conjunto das features mais significativas a determinados conjuntos de aplicações candidatas. Cada conjunto denota a existência de um subdomínio diferente. Dessa maneira, a partir de uma LPF podem ser derivados: (I) frameworks bem restritos conforme os requisitos das aplicações ou se houver necessidade, (II) frameworks que englobam um subdomínio inteiro coberto pelo framework.

Assim, a modularização de Frameworks de Múltiplos Domínios em LPFs é uma alternativa que além de permitir melhores níveis de abstração, permite alcançar

melhorias quanto à flexibilidade de composição de features, produtividade com a redução da tendência a erros no processo de instanciação desses frameworks e o tratamento de features que são dispensáveis.

9.1 Contribuição e Síntese dos Principais Resultados

A contribuição do trabalho possui três vertentes: (a) refinamentos dos conceitos de LPF, (b) uma abordagem de modularização de Frameworks de Múltiplos Domínios em LPF, considerando esses refinamentos, (c) resultados de uma avaliação empírica que avalia a modularização dos subdomínios de um FMD no sentido de obter LPFs e resultados de um estudo comparativo de algumas aplicações criadas a partir da instanciação de duas LPFs, considerando os cenários de utilização, comentados anteriormente e do FMD precursor.

A abordagem é voltada para frameworks que apresentam dificuldades quanto ao gerenciamento de suas features e inflexibilidade arquitetural. Quanto aos refinamentos dos conceitos de LPF, apresentou-se uma exploração das diferenças relacionadas ao modelo e ao comportamento das features em uma LPF, comparando com as LPS convencionais; descreveu-se os possíveis cenários de utilização de uma LPF, em que dependendo do propósito de uma LPF, implica-se em definir uma forma diferente de modularização. Com relação ao experimento, apesar dos princípios de uma LPF serem independentes de uma linguagem de implementação específica, conduziu-se uma avaliação verificando o tempo consumido para modularizar um FMD em LPFs e os erros encontrados durante esse processo, considerando as técnicas POA com a linguagem AspectJ e MDD com templates Acceleo. Vale enfatizar que o tempo dispendido para realizar a modularização não é tão importante quanto à qualidade da LPF obtida. Como resultados, tanto o tempo consumido para a modularização quanto à quantidade de erros, são aproximadamente equivalentes considerando testes estatísticos.

De modo geral, constatamos que a modularização de frameworks em LPF, permite solucionar dificuldades quanto ao gerenciamento de features, o que inclui evitar a presença de features desnecessárias do framework nas aplicações finais e obter melhorias em termos de manutenção e evolução que são indispensáveis para

a escalabilidade de um framework. Além disso, verificamos que por meio do tratamento das features em uma LPF, é possível eliminar variabilidades desnecessárias em um membro derivado da linha. Por exemplo, métodos abstratos que estejam disponíveis na instanciação do framework, mas que são específicos para o reuso de features de domínios diferentes de uma aplicação em desenvolvimento. Assim, a tendência em cometer erros por parte do Engenheiro de Aplicação durante a instanciação do framework também pôde ser reduzida.

9.2 Limitações

A principal limitação quanto à abordagem de modularização de frameworks em LPF é que ainda não existe uma ferramenta completa que apoie sequencialmente o processo de modularização apresentado. Com uma ferramenta completa, poderemos investigar o impacto da inserção de novas features na arquitetura de uma LPF, analisando e tratando as possíveis interferências. Além disso, essa ferramenta também poderá auxiliar o processo de escolha de features em vários níveis. Por exemplo, a partir de uma LPF além de compor um membro com um subconjunto das features que são fornecidas, em um nível posterior, considerando esse subconjunto, poderíamos derivar um membro ainda mais restrito segundo os requisitos das aplicações que se deseja desenvolver.

9.3 Trabalhos Futuros

Considerando os conceitos apresentados, uma das perspectivas futuras do trabalho consiste em aplicar o processo *FMDtoLPF* em outros frameworks, realizando outros estudos de caso. Pretende-se explorar a possibilidade do desenvolvimento de uma LPS a partir de uma LPF proveniente de um framework de aplicação, incluindo as classes que o instanciam e com isso compor e testar as aplicações derivadas dessa linha.

Além dessa perspectiva, pretende-se explorar a capacidade de uma LPF ser mantida e utilizada sob os conceitos dos Ecossistemas de Software (SECOS) (Jansen e Cusumano, 2012), visto que uma LPF pode ser constituída por diversos desenvolvedores em uma plataforma *open-source* e distribuída, isto é, uma rede colaborativa. Desse modo, possibilita-se a criação de LPFs flexíveis e orientadas aos domínios dos negócios. Uma LPF poderá ser disponibilizada para que outros desenvolvedores possam adicionar features ou melhorar as existentes. Com isso, muitos frameworks que são difíceis de serem encontrados, geralmente frameworks de aplicação que ficam restritos como software proprietário, poderão ser fornecidos e melhorados no contexto de SECOS.

Pretendemos investigar também a aplicação da abordagem apresentada em outros frameworks, a fim de verificarmos o quão produtivo torna-se a evolução das LPFs obtidas em comparação com seus frameworks precursores. Além disso, pretendemos utilizar outras tecnologias para compararmos o esforço necessário na modularização de frameworks em LPFs.

Em um momento futuro, também pretendemos estender a proposta de concepção de uma LPF centralizada a partir de vários frameworks relacionados. Acredita-se que a criação de um *plug-in* para visualização do mapeamento entre features x classes, mostrando quais classes implementam uma determinada feature, e quais classes são afetadas caso uma feature seja selecionada, possa auxiliar o EF na modularização de seus frameworks. Além desse *plug-in*, pretendemos desenvolver uma ferramenta que permita a criação de membros de uma LPF em vários níveis. No primeiro nível, o EF cria um membro contendo as features de um determinado domínio, em seguida, em um segundo nível, se julgar necessário, seleciona um conjunto mais específico de features que façam parte desse subdomínio.

Além dessas possibilidades de extensão do trabalho, pretende-se investigar como os testes de software, especificamente: teste de fluxo de dados, teste estrutural e teste de mutação, que são aplicados em LPS poderiam ser empregados no contexto de LPF. A partir disso, desenvolver um conjunto de diretrizes que auxilie a aplicação desses testes em LPF e assim, obter membros com mais qualidade, solucionando problemas quanto ao seu reuso e quanto à arquitetura/modularidade das features.

REFERÊNCIAS

ACCELEO. User Guide. Disponível em: <http://www.eclipse.org/acceleo/>. Acesso em: 23 de Setembro 2012.

ARES C., J.M.; DIESTE Tubio, O.; GARCIA Vazquez, R.; LOPEZ Fernandez, M.; RODRIGUEZ Yanez, S.; Formalising the Software Evaluation Process. Proceedings In 18th International Conference of the Chilean Society of Computer Science, Washington, DC, USA, pp15 – 24, 1998.

BARREIROS, J., Moreira, A. Soft Constraints in Feature Models. In Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA), pp.136-141, 2011.

BASS, L.; Clements, P.; KAZMAN, R.. Software Architecture in Practice, 2nd Edition.[S.l.]: Addison-Wesley, 2003.

BATORY D., LOFASO B., e SMARADAKIS Y.. JTS: Tools for Implementing Domain-Specific Languages, 5th International Conference on Software Reuse, Victoria, Canada, 1998.

BATORY D., CARDONE, R., SMARADAKIS, Y.. “Object-oriented framework and product lines”. In Proceedings of the first conference on Software product lines: experience and research directions, pp. 227-247, 2000.

BATORY D., SHEPHERD C.T.. “Product Lines of Product Lines”. Technical Report University of Texas, Department of Computer Science. Disponível em: <http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2049.pdf>. Acesso em: 20 de novembro de 2011.

BATORY D., LOPES-HERREJON R. E., MARTIN J.P. (2002). “Generating Product-Lines of Product-Families”. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE’02), 2002

BENAVIDES D., P. TRINIDAD e RUIZ-CORTÉS A.. "Automated Reasoning on Feature Models". 17th Conference on Advanced Information Systems Engineering (CAiSE'05). Porto, Portugal. 2005.

BOSCH, J.; MOLIN, P.; MATTSSON, M.; BENGTSSON, P.; FAYAD, M.. Framework problem and experiences in Application Frameworks: Object-Oriented Foundations of Framework Design, John Willey and Sons, p. 55–82, 1999.

BRACHA e COOK W., “Mixin-Based Inheritance”, ECOOP/OOPSLA, pg 303-311, 1990.

BRAGA, R. T. V., GERMANO, F. S. R., MASIERO, P. C... A Pattern Language for Business Resource Management. In Proceedings of the 6th Patttern Language of Programs Conference, pp.1–33, 1999.

BRAGA, R. T. V.. Um processo para Construção e Instanciação de Frameworks baseados em uma Linguagem de Padrões para um Domínio Específico. Tese de doutorado. Instituto de Ciências Matemáticas e Computação (ICMC-USP), Universidade de São Paulo, Dezembro, 2002.

BORGES, V.; GARCIA R.; VALENTE, M. T.. Uma Ferramenta para Extração Semi-automática de Linhas de Produtos de Software Usando Coloração de Código. I Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas), p. 73-78, 2010.

BUSHMANN, F. et at. A System of Patterns, Wiley, 1996.

CLEMENTS, P., NORTHROP, L.. Software Product Lines: Practices and Patterns. Addison Wesley. 2002.

CODENIE W., HONDT K., STEYAERT P., VERSAMMEN A.. From Custom Applications To Domain-Specific Frameworks. Communications of the ACM, pg. 70-77, 1997.

COPLIEN J., HOFFMAN D., e WEISS D.. Commonality and Variability in Software Engineering. IEEE Computer Society, pg. 37 - 45, 1999.

CZARNECKI K. e EISNECKER U.W.. Generative Programming: Methods, Tools, and Applications. Addison Wesley, 2000.

CZARNECKI K., HELSEN S. e ULRICH E.. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Journal Software Process: Improvement and Practice. New York, USA, pg. 143 – 169, 2005.

DIJKSTRA, E. W. A Discipline of Programming. Prentice-Hall, 1976.

DURELLI, V. H. S., DURELLI, R. S., BRAGA, R. T. V., BORGES, S. S.. A Domain Specific Language for Lessening the Effort Needed to Instantiate Applications Using GRENJ Framework. In Information Systems Brazilian Symposium, pp. 31-40, 2010.

EZRAN, M.; MORISIO, M.; TULLY, C.. Practical Software Reuse. [S.l.]: Springer, 2002.

FAYAD, M. E.. JOHNSON, R. E.. Domain-specific application frameworks: frameworks experience by industry. New York, USA: J. Wiley, 2000.

FAYAD, M. E., SCHMIDT, D. C.. Object-Oriented Application frameworks. Communications". ACM, Vol. 40, 10 p.32-38. New York, USA, 1997.

FAYAD, M. E.; SCHMIDT; D.C.,JOHNSON, R. E.. Implementing Application frameworks: object-oriented frameworks at work. New York: J. Wiley, 1999a.

FAYAD, M. E., SCHMIDT; D. C., JOHNSON, R. E.. Building application frameworks: object-oriented foundations of framework design". New York, USA: J. Wiley, 1999b.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: a research roadmap. In: Proceedings of the 29th International Conference on Software Engineering - Future of Software Engineering, Minneapolis, USA, p. 37-54, 2007.

GALL, H. C.; KLÖSH, R. R.; MITTERMEIER, R. T.. Application patterns in reengineering: Identifying and using reusable concepts. In: 6th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, p. 1099–1106, 1996

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.. Design patterns - abstraction and reuse of object-oriented design. Lecture Notes on Computer Science, n. 707, p. 406–431, 1993.

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

GOSSET, W. S. "Student's" Collected Papers. Biometrika Office, 1947.

GOVONI, D.. Java Application Frameworks. John Wiley & Sons, 1999.

GRONBACK, R. C. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, 2009.

HAZEN, A., Storage to be provided in impounding reservoirs for municipal water supply, Trans. Am. Soc. Civ. Eng., 77, 1539 –1640, 1914.

JANSEN, S., CUSUMANO, M... Defining Software Ecosystems: A Survey of Software Platforms and Business Network Governance. In Proceedings of the 4th Workshop on Software Ecosystem, Boston, MA, USA, 2012

JOHNSON, R. E. Reusing Object-Oriented Design. University of Illinois, Technical Report UIUCDCS 91-1696, 1991.

JOHNSON, R. E.. Frameworks = (components + patterns). Communications of the ACM, v. 40,n. 10, p. 39–42, 1997.

JOHNSON R. E.. FOOTE B.. Designing Reusable Classes. In Journal of Object-Oriented Programming, Vol.1, nº 2, p. 22-35. Disponível em: http://www.laputan.org/drc/drc.html, 1988.

JOHNSON, R. E.. Reusing Object-Oriented Design. University of Illinois, Technical Report UIUCDCS 91-1696, 1991.

PARREIRA Junior, P.A. MENDES, W.; de CAMARGO, V.V. ; PENTEADO, R.A.D.; COSTA, H.A.X.. Mining crosscutting concerns with ComSCId: A rule-based customizable mining tool. XXXVIII Conferencia Latino-americana em Informática (CLEI), pg. 1-9, 2012.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E, PETERSON, A. S.. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.

KÄSTNER, C., APEL, S., KUHLEMANN, M.. Granularity in Software Product Lines. Proceeding ICSE '08 Proceedings of the 30th international conference on Software Engineering. New York, NY, USA, 2008.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., IRVING, J.. Aspect Oriented Programming. In: Proceedings of 11 ECOOP, 1997.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., GRISWOLD, G. Getting Started With AspectJ, Communications of the ACM, vol 44, No. 10, pp.59-65, 2001.

KLEPPE, A.; WARMER, J.; BAST, W. MDA explained: the model driven architecture: practice and promise. Boston: Pearson Education, 2003.

KRUEGER, C. W.. Software Reuse, ACM Computing Surveys, vol. 24, Nº. 02, June, pp. 131-183, 1992.

KULESZA, U., SANT'ANNA, C., LUCENA, C.. Refactoring the JUnit Framework using Aspect-Oriented Programming. Conference on Object-Oriented Programming Systems Languages and Applications. pg. 136-137, 2005.

LADDAD, R.. AspectJ in Action: Practical Aspect-Oriented Programming, Manning, Greenwich, 2003.

LOESCH, F., PLOEDEREDER, E.. Optimization of Variability in Software Product Lines. In Proceedings of the Software Product Line Conference, SPLC, pg. 151-162, 2007.

LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. Tese de Doutorado. Universidade de São Paulo. Instituto de Ciências Matemáticas e de Computação, São Carlos, SP, 2009.

MONTGOMERY, D. C. Design and Analysis of Experiments. 5 ed., Wiley, 2000.

NAUR, P. Revised Report on the Algorithmic Language ALGOL 60. Communications of the ACM, Vol. 3 No.5, pp. 299-314, 1960.

OLIVEIRA, A.L., FERRARI, F.C., PENTEADO, R.A.D., CAMARGO, V. V.. Investigating Framework Product Lines. In: ACM Symposium on Applied Computing, 27th ACM Symposium on Applied Computing, 2012.

OLIVEIRA JUNIOR, E. A.; GIMENES, I. M. S.; MALDONADO, J. C. Systematic Management of Variability in UML-based Software Product Lines. Journal of Universal Computer Science, v. 16, n. 17, p. 2374-2393, 2010.

OMG. MDA guide version 1.0.1. OMG: 2003. Disponível em <http://www.omg.org/docs/omg/03-06-01.pdf>. Acesso em: 25 maio 2013.

OMG. Meta Object Facility Core Specification Version 2.0. 2006. Disponível em: <http://www.omg.org>. Acesso em: 25 de maio de 2013.

OMG. MOF Model To Text Transformation Language (MOFM2T), 1.0. Disponível em <http://www.omg.org/spec/MOFM2T/1.0/PDF>. Acesso em: 22 maio 2013

PARSONS, D., RASHID, A. ; SPECK, A. ; TELEA, A. A “framework” for object oriented frameworks design. Proceedings of Technology of Object-Oriented Languages and Systems, 1999, pg 141 – 151, 1999.

PINTO, S. C. C. S.. Composição em Web Frameworks, tese de doutorado, Departamento de Informática PUC-Rio, 2000.

PREE, W., KOSHIMIES K.. Framelets – Small and Loosely Coupled Frameworks. ACM Computer Surveys, v.32, n.1, pp.6-10, 2000.

PREE, W.. Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.

PRESSMAN, Roger S. Engenharia de Software. Mcgraw-hill Interamericana 6ª ed., 2006.

REENSKAUG, et al., OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems, Journal of Object-Oriented Programming, pg. 27-41, 1992.

ROGERS G.. Framework-Based Software Development in C++. Prentice-Hall, 1997.

ROSS, D.. Structured Analysis: A Language for Communicating Ideas. IEEE Transaction on Software Engineering, v.3, n.1, 1977."

SCHMID, H.A.. Systematic Framework Design by generalization. Communications of the ACM, V. 40, nº 10, p. 48-51.New York, USA, 1997.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. IEEE Computer, v. 39, n. 2, p. 25-31, 2006.

SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C.. Feature Diagrams: A Survey and a Formal Semantics, Requirements Engineering, 14th IEEE International Conference, pg.139-148, 2006.

SMARAGDAKIS e BATORY D., Implementing Layered Designs with Mixin Layers", ECOOP 1998.

SOMMERVILLE, I. (2011). Software Engineering. Addison Wesley/Pearson Education - International Version, 9th edition.

STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. EMF – Eclipse Modeling Framework, Addison-Wesley, 2008.

SHAPIRO, S.S. & WILK, M.B. An analysis of variance test for normality (complete samples). Biometrika, 52:591-611, 1965.

SZYPERSKI, C.. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1997.

THÜM Thomas, KÄSTNER Christian, BENDUHN Fabian, MEINICKE Jens, SAAKE Gunter e LEICH Thomas. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. Science of Computer Programming, 2013.

TRAVASSOS, G. H. ; GUROV, D.; AMARAL, E. A. G. Introdução à Engenharia de Software Experimental. Relatório Técnico RT-ES-590/02, Programa de Engenharia de Sistemas e Computação, UFRJ, 2002."

VANHILST e NOTKIN D.. Using C++ Templates to Implement Role-Based Designs, JSSST International Symposium on Object Technologies for Advanced Software, Springer-Verlag, 1996, 22-37

WILCOXON, F. Individual Comparisons by Ranking Methods. Biometrics Bulletin 1: 80–83, 1945.

WINCK D. V. e JUNIOR V. G.. AspectJ – Programação Orientada a Aspectos com Java. Novatec, 2006.

WOHLIN, C.; RUNESON, P.; HÖST, M.; REGNELL, B.; WESSLÉN A.. “Experimentation in Software Engineering: an Introduction”. Kluwer Academic Publishers, Norwer, MA, USA, 2000.

YASSIN, A.; FAYAD, M. E.. Application frameworks: A survey, cáp. 29 in M. E. Fayad and R. E. Johnson. Domain-Specific Application Frameworks: Frameworks Experience by Industry, John Wiley & Sons, p. 615–632, 2000.

Apêndice A

FORMULÁRIO DE CONSENTIMENTO

Título do projeto e Propósito

Experimento comparativo entre as Versões: Orientada a Objetos modularizada com Features/Templates e Orientada a Aspectos do Framework GRENJ

O objetivo do experimento é comparar o tempo necessário para realizar evoluções com as versões OO com features/templates e OA do framework GRENJ, que é um framework de aplicação baseado na linguagem de padrões GRN. A partir da comparação serão verificadas as vantagens e desvantagens entre as duas versões do GRENJ.

Declaração de idade
Eu declaro que sou maior de 18 anos de idade e que desejo participar do experimento conduzido por Victor Hugo Santiago Costa Pinto, estudante de mestrado do PPGCC-UFSCar, no contexto da disciplina de Tópicos em Engenharia de Software.

Procedimentos
O experimento envolve três etapas. Na primeira etapa será conduzido um treinamento que aborda a realização de evoluções nas versões OO com templates/features e OA do framework GRENJ. Na segunda etapa será realizado um experimento piloto com dois cenários de evolução. Em cada cenário deverá ser utilizado as duas versões do GRENJ. Na terceira etapa serão utilizados outros dois cenários e os procedimentos para das duas versões do framework serão similares em termos de esforço necessário aos utilizados na segunda etapa.

Confidência
Todas as informações coletadas no experimento são confidenciais, e meu nome não será identificado em tempo algum.

Benefícios, liberdade para retirar-se.
Estou consciente que não terei remuneração pela participação no experimento, mas que o estudo ajudará a aprender mais sobre o uso de frameworks de aplicação OO e OA. Eu entendo que sou livre para perguntar ou para retirar minha participação em

qualquer tempo sem penalidade, e que eu terei acesso aos resultados principais do experimento.

Responsáveis

Victor Hugo S. C. Pinto
Univ. Federal de São Carlos
São Carlos-SP

Prof. Dr. Valter V. de Camargo
Univ. Federal de São Carlos
São Carlos-SP

Prof. Dra. Rosângela A. D. Penteado
Univ. Federal de São Carlos São Carlos-SP

Nome do participante

Assinatura do participante Data

APÊNDICE B

FORMULÁRIO DE CATEGORIZAÇÃO DOS PARTICIPANTES

Este formulário teve por objetivo caracterizar o perfil de cada participante, quanto às suas experiências acadêmicas e profissionais, considerando os requisitos do experimento. Esse formulário foi disponibilizado por meio do link: http://goo.gl/Wwo5Z, com os seguintes questionamentos:

1) Conhecimento dos conceitos de Orientação a Objetos:

( ) Alto.

( ) Médio.

( ) Baixo.

2) Nível de conhecimento em relação à linguagem Java:

( ) Básico – aproximadamente 1 ano.

( ) Intermediário – mais de 2 anos.

( ) Avançado – 5 anos ou mais.

3) Conhecimento dos conceitos de Orientação a Aspectos (interesses, *pointcuts*, *advices*, *inter-type declarations*, designadores *this* e *target*):

( ) Alto.

( )Médio.

( ) Baixo (somente o que foi apresentado nesta disciplina).

4) Conhecimento dos conceitos de Templates (regras para geração de código):

( ) Alto.

( ) Médio.

( ) Baixo (somente o que foi apresentado nesta disciplina).

5) Classifique sua experiência em sobrescrever métodos abstratos em Java:

( ) Tenho alguma experiência.

( ) Tenho experiência intermediária.

( ) Tenho bastante experiência.

6) Nível de conhecimento em relação à linguagem AspectJ:

( ) Baixo (somente o que foi apresentado nesta disciplina).

( ) Intermediário – mais de 1 ano.

( ) Avançado – 2 anos ou mais.

7) Em qual das categorias abaixo você se encaixa como desenvolvedor?

( ) sistemas triviais desenvolvidos durante o curso de graduação.

( ) sistemas de médio porte; com aproximadamente 7-20 mil linhas de código.

( ) sistemas de grande porte; com aproximadamente 20-50 mil linhas de código.

8) Classifique seu conhecimento em relação ao framework GRENJ:

( ) Pouco (somente o que foi apresentado nesta disciplina).

( ) Intermediário (teve contato com o GRENJ fora desta disciplina).

( ) Avançado.

9) Classifique seu conhecimento em relação à linguagem de padrões GRN:

( ) Pouco (somente o que foi apresentado nesta disciplina).

( ) Intermediário (teve contato com a GRN fora desta disciplina).

( ) Avançado.

10) Classifique seu nível de habilidade com o ambiente Eclipse:

( ) Tenho alguma experiência.

( ) Tenho experiência intermediária.

( ) Tenho bastante experiência.

.

APÊNDICE C

ROTEIRO DE ATIVIDADES E FORMULÁRIO DE COLETA DE DADOS

Grupo 1: Manual para a realização das tarefas de Evolução

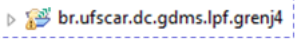
Nome:

Evolução 1 com template/features

Externalizar a feature Trade como uma feature opcional do framework

1º) Definir as classes que implementam a **feature Trade** (utilize a tabela)

2º) Criar uma **condição** para Trade no template principal **main.mtl** e vincular os geradores das classes (definidas no 1º passo) a essa condição



3º) Descobrir se existe outras classes que dependem das classes da feature Trade
-> Se existir dependências (utilize a tabela):
Deve-se anotar com IF('Trade') essas dependências (atributos, métodos e refinamentos) para que sejam introduzidas se a **feature Trade** for requerida

Para testar: Exporte o projeto que tem os templates e gere a partir do arquivo de configuração **(.config)**, um membro do framework com apenas as **features GRENJ** e **Rental**.
-Verifique se o membro gerado possui erros, copie o membro para Evolução 1 e execute o clean no projeto Evolução1 !

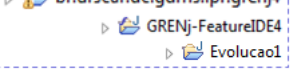


Tabela para marcação do tempo

Etapas	Tempo Inicial (h:min)	Tempo Final (h:min)
1ª		
2ª		
3ª		
Teste		

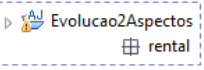
Erros:

Evolução 2 com Aspectos

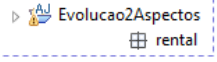
Tornar a feature Rental uma feature opcional do framework

1º) Definir as classes que implementam a **feature Rental** (utilize a tabela)

2º) Mover para o pacote **rental** as classes (definidas no 1º passo)



3º) Descobrir se existe outras classes que dependem das classes da feature Rental
-> Se existir dependências (utilize a tabela):
Deve-se modificar o aspecto 'MeuAspecto' dentro do pacote **rental** para introduzir essas dependências (atributos, métodos e refinamentos) nas classes dependentes, caso a **feature Rental** seja requerida



Para testar: Copie o pacote **rental** com as classes + aspectos para o projeto Evolucao2Troca;
Remova o pacote **rental do projeto Evolucao2Aspectos**
Verifique se o projeto **Evolucao2Aspectos** no qual você apagou o pacote **rental** continua integro e sem erros (execute o clean)

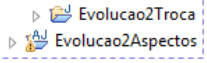


Tabela para marcação do tempo

Etapas	Tempo Inicial (h:min)	Tempo Final (h:min)
1ª		
2ª		
3ª		
Teste		

Erros:

Grupo 2: Manual para a realização das tarefas de Evolução

Nome:

Evolução 2 com template/features

Externalizar a feature Rental como uma feature opcional do framework

1º) Definir as classes que implementam a **feature Rental** (utilize a tabela)

2º) Criar uma **condição** para Rental no template principal **main.mtl** e vincular os geradores das classes (definidas no 1º passo) a essa condição

br.ufscar.dc.gdms.lpf.grenj4

3º) Descobrir se existe outras classes que dependem das classes da feature Rental
-> Se existir dependências (utilize a tabela):
Deve-se anotar com IF('Rental') essas dependências (atributos, métodos e refinamentos) para que sejam introduzidas se a **feature Rental** for requerida

br.ufscar.dc.gdms.lpf.grenj4

Para testar: Exporte o projeto que tem os templates e gere a partir do arquivo de configuração **(.config)**, um membro do framework com apenas as **features GRENJ** e **Trade**.
-Verifique se o membro gerado possui erros, copie o membro para Evolução 2 e execute o clean no projeto Evolução2 !

br.ufscar.dc.gdms.lpf.grenj4
GRENj-FeatureIDE4
Evolucao2

Tabela para marcação do tempo

Etapas	Tempo Inicial (h:min)	Tempo Final (h:min)
1ª		
2ª		
3ª		
Teste		

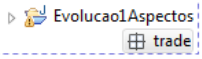
Erros:

Evolução 1 com Aspectos

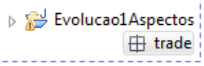
Tornar a feature Trade uma feature opcional do framework

1º) Definir as classes que implementam a **feature Trade** (utilize a tabela)

2º) Mover para o pacote **trade** as classes (definidas no 1º passo)



3º) Descobrir se existe outras classes que dependem das classes da feature Trade
-> Se existir dependências (utilize a tabela):
Deve-se modificar o aspecto 'MeuAspecto' dentro do pacote **trade** para introduzir essas dependências (atributos, métodos e refinamentos) nas classes dependentes, caso a **feature Trade** seja requerida



Para testar: Copie o pacote **trade** com as classes + aspectos para o projeto Evolucao2Troca;
Remova o pacote **trade do projeto Evolucao1Aspectos**
Verifique se o projeto **Evolucao1Aspectos** no qual você apagou o pacote **trade** continua integro e sem erros (execute o clean)

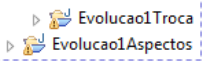


Tabela para marcação do tempo

Etapas	Tempo Inicial (h:min)	Tempo Final (h:min)
1ª		
2ª		
3ª		
Teste		

Erros:

APÊNDICE D

FORMULÁRIO DE OPINIÃO, DIFICULDADES E SUGESTÕES

Este formulário teve por objetivo caracterizar as impressões dos participantes a cerca da condução do experimento até o momento da execução do piloto, visando obter suas opiniões, dificuldades encontradas e as sugestões para melhoria do experimento final. Esse formulário foi disponibilizado por meio do link: http://goo.gl/TFTL2. Neste formulário, havia somente três questionamentos:

1) Caso você teve dificuldades na execução do piloto, quais foram?
2) Como você avalia o treinamento e qual sua opinião sobre o experimento até o momento?
3) O que você sugere como melhorias para o experimento?

APÊNDICE E

FORMULÁRIO FINAL

Este documento teve por objetivo caracterizar a visão dos alunos com relação à facilidade na utilização das duas tecnologias (Programação Orientada a Aspectos com AspectJ e Desenvolvimento Dirigido a Modelos com templates Acceleo), em qual delas obteve-se maior quantidade de erros e qual a sugestão para minimizar os erros obtidos. Esse formulário foi aplicado após o experimento real e foi fornecido por meio do link: http://goo.gl/4y3iP. Os itens desse formulário foram os seguintes:

1) Quanto a facilidade de realizar evoluções no framework GRENJ, indique o grau de facilidade nas duas tecnologias.

	1	2	3	4	5
Orientação a Aspectos (AspectJ)					
Templates/Features (Acceleo)					

2) Considerando sua resposta na pergunta 1, em qual tecnologia você obteve mais erros?

( ) Orientação a Aspectos (AspectJ)

( ) Templates/Features (Acceleo)

( ) Não tive erros

3) Considerando sua resposta na pergunta 2, qual sua sugestão para minimizar os erros obtidos?

Anexo A

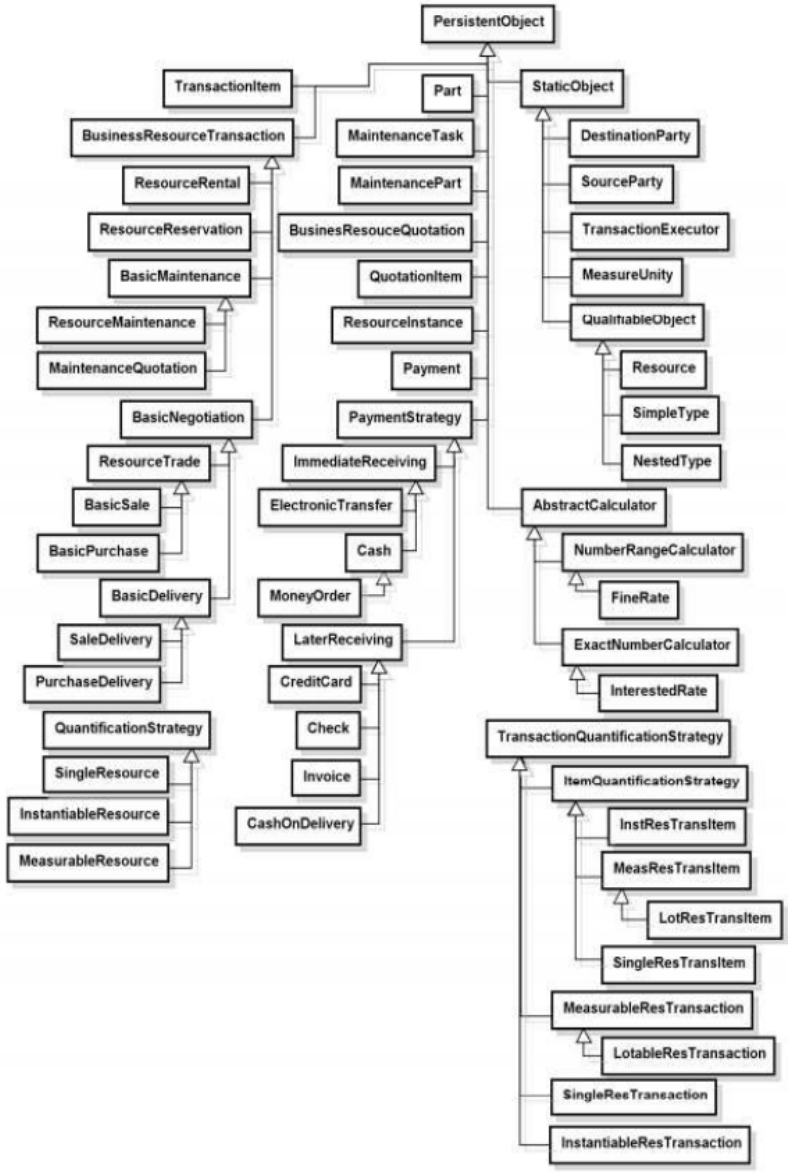
TABELA PADRÃO DE DISTRIBUIÇÃO DE PROBABILIDADE ESTATÍSTICA *T* DE *STUDENT**

Área contida nas duas caudas laterais (bicaudal) da distribuição *t* de *Student*									
gl/α		**Graus de Significância**							
		0,9900	**0,9500**	**0,9000**	**0,1000**	**0,0500**	**0,0275**	**0,0100**	**0,0050**
Graus de Liberdade	**1**	0,0157	0,0787	0,1584	6,3138	12,7062	23,1354	63,6567	127,3213
	2	0,0141	0,0708	0,1421	2,9200	4,3027	5,9051	9,9248	14,0890
	3	0,0136	0,0681	0,1366	2,3534	3,1824	4,0281	5,8409	7,4533
	4	0,0133	0,0667	0,1338	2,1318	2,7764	3,3912	4,6041	5,5976
	5	0,0132	0,0659	0,1322	2,0150	2,5706	3,0790	4,0321	4,7733
	6	0,0131	0,0654	0,1311	1,9432	2,4469	2,8954	3,7074	4,3168
	7	0,0130	0,0650	0,1303	1,8946	2,3646	2,7749	3,4995	4,0293
	8	0,0129	0,0647	0,1297	1,8595	2,3060	2,6899	3,3554	3,8325
	9	0,0129	0,0645	0,1293	1,8331	2,2622	2,6269	3,2498	3,6897
	10	0,0129	0,0643	0,1289	1,8125	2,2281	2,5782	3,1693	3,5814
	11	0,0128	0,0642	0,1286	1,7959	2,2010	2,5396	3,1058	3,4966
	12	0,0128	0,0640	0,1283	1,7823	2,1788	2,5082	3,0545	3,4284
	13	0,0128	0,0639	0,1281	1,7709	2,1604	2,4821	3,0123	3,3725
	14	0,0128	0,0638	0,1280	1,7613	2,1448	2,4602	2,9768	3,3257
	15	0,0127	0,0638	0,1278	1,7531	2,1314	2,4414	2,9467	3,2860
	16	0,0127	0,0637	0,1277	1,7459	2,1199	2,4252	2,9208	3,2520
	17	0,0127	0,0636	0,1276	1,7396	2,1098	2,4111	2,8982	3,2224
	18	0,0127	0,0636	0,1274	1,7341	2,1009	2,3987	2,8784	3,1966
	19	0,0127	0,0635	0,1274	1,7291	2,0930	2,3877	2,8609	3,1737
	20	0,0127	0,0635	0,1273	1,7247	2,0860	2,3778	2,8453	3,1534
	21	0,0127	0,0635	0,1272	1,7207	2,0796	2,3690	2,8314	3,1352
	22	0,0127	0,0634	0,1271	1,7171	2,0739	2,3610	2,8188	3,1188
	23	0,0127	0,0634	0,1271	1,7139	2,0687	2,3538	2,8073	3,1040
	24	0,0127	0,0634	0,1270	1,7109	2,0639	2,3472	2,7969	3,0905

* “Student's” Collected Papers (Ed. Egar S. Pearson e J. Wishart, University College, Londres, 1942).

Anexo B

HIERARQUIA DAS CLASSES DA CAMADA MODELO DO FRAMEWORK GRENJ**



** Diagrama fornecido no Cook book do GRENJ (figura 7.1), desenvolvido pelo grupo de pesquisa *AdvanSE* (do Departamento de Computação da UFSCar), em setembro de 2012.