UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS LAZZARI TOMASI

Ferramenta para Recomendação de Configuração de Modelos de *Features* com base em Preferências de Múltiplos *Stakeholders*

Trabalho de Conclusão apresentado como
requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Profa. Dra. Ingrid Oliveira de Nunes
Orientador

Porto Alegre, julho de 2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Lazzari Tomasi, Lucas

Ferramenta para Recomendação de Configuração de Modelos de *Features* com base em Preferências de Múltiplos *Stakeholders* / Lucas Lazzari Tomasi. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2015.

42 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2015. Orientador: Ingrid Oliveira de Nunes.

1. Linhas de produto de software. 2. Modelo de features. 3. Configuração de produtos. 4. Escolha social. I. Oliveira de Nunes, Ingrid. II. Título.

“DON’T
PANIC”
— DOUGLAS ADAMS

AGRADECIMENTOS

Agradeço ao apoio da minha família durante todos esses anos, principalmente ao meu irmão Diego, meu pai Sergio e minha tia Rosa.

Agradeço aos amigos que fiz ao longo do curso pela ajuda no aprendizado em todos os semestres e aos demais colegas que de alguma maneira fizeram parte deste percurso.

Agradeço à minha orientadora Ingrid pela grande ajuda no desenvolvimento deste trabalho, estando sempre presente e à disposição. Também agradeço aos demais membros do grupo Prosoft pelas contribuições oferecidas.

Finalmente gostaria de dedicar este trabalho à minha mãe Juraci, que infelizmente não pode estar conosco até o final dessa jornada, e sem ela nada disso teria sido possível.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

LPS	Linha de Produto de Software
IDE	Integrated Development Environment
AVG	Average
ANP	Average Number of Preferences
BC	Borda Count
CR	Copeland Rule
LM	Least Misery
MP	Most Pleasure
MUL	Multiplicative
FOSD	Feature-Oriented Software Development
XML	eXtensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Linhas de Produto de Software (LPS) surgiram como uma nova maneira de produzir software com reúso mais organizado e em larga escala. Modelos de *features* são usados para representar e organizar a variabilidade em uma LPS. Portanto, a tarefa de configurar um modelo de *features* de modo a gerar uma configuração de produto válida torna-se uma atividade chave. Essa tarefa, conhecida por ser difícil, propensa a erros e demorada, torna-se ainda mais complicada quando múltiplos *stakeholders* estão envolvidos no processo de configuração. A ferramenta apresentada neste trabalho tem o propósito de dar suporte ao processo de configuração com múltiplos *stakeholders*, recomendando configurações ótimas baseadas em suas preferências através de um plug-in para o Eclipse. Para isso foram utilizadas sete estratégias baseadas na teoria da escolha social visando alcançar uma maior satisfação entre os *stakeholders*.

Palavras-chave: Linhas de produto de software, modelo de features, configuração de produtos, escolha social.

Feature Model Configuration Recommendation Tool based on Multiple Stakeholders Preferences

ABSTRACT

Software Product Lines (SPL) came as a new way of producing software with a large-scale and more organized reuse. Feature models are used to represent and organize the variability in a SPL. Therefore, the task of configuring a feature model in order to generate a valid product configuration becomes a key activity. This task, which is known to be hard, error-prone and time consuming, gets even harder when many stakeholders are involved in the process of configuration. The tool presented in this paper has the purpose of supporting the configuration process with multiple stakeholders, recommending optimal configurations based on the their preferences via an Eclipse plug-in. To do so we used seven strategies based on social choice theory aiming for a greater satisfaction among the stakeholders.

Keywords: software product lines, feature model, product configuration, social choice.

1 INTRODUÇÃO

O reúso de componentes não é um conceito novo, é uma das estratégias usadas para a redução de custos e esforço no desenvolvimento de sistemas de software. As linhas de produto surgiram com esta motivação, sistematizando o reúso, criando famílias de produtos com características comuns entre si, permitindo um aumento de produção, mas com a possibilidade de customização. Na indústria de software a combinação dos conceitos de famílias de produtos e customização em massa deram origem às chamadas Linhas de Produto de Software (LPS). Por meio da organização e sistematização do reúso é possível diminuir custos de desenvolvimento, melhorar a qualidade do software e reduzir o tempo de produção.

Nas Linhas de Produto de Software, o uso de modelos de *features* destaca-se como uma forma de representar e gerenciar a variabilidade dentro de uma LPS. O modelo de *features* é uma representação gráfica em forma de árvore que permite visualizar as *features* disponíveis de serem selecionadas de forma a criar um novo produto customizado, assim sendo, ele modela as propriedades comuns e variáveis dos produtos possíveis de uma LPS, incluindo suas interdependências.

A partir de um modelo de *features* definido é possível a criação de configurações de produtos. Uma configuração de um modelo de *features* (i.e. configuração de um produto) é a seleção de um conjunto de *features* contidas nesse modelo. Dado que os modelos de *features* podem ter um grande número de *features* disponíveis, temos um número exponencial de configurações possíveis. Desta forma, escolher um conjunto válido de *features*, satisfazendo as restrições do modelo e as preferências de múltiplos *stakeholders* torna-se uma tarefa nada trivial, exigindo suporte computacional.

Para chegar na definição de uma configuração de um produto é comum termos opiniões de múltiplas partes envolvidas no processo. Quando há um número maior de indivíduos envolvidos no processo de configuração de um mesmo produto torna-se complicado de encontrar um consenso para determinar quais *features* devem estar presentes no produto final, de forma que os participantes do processo fiquem suficientemente satisfeitos. Nesta abordagem consideramos um cenário onde vários *stakeholders* envolvidos no processo de configuração de um produto podem expressar suas opiniões e preferências. *Stakeholders* podem ser qualquer indivíduo que tem um interesse no domínio, como por exemplo: gerentes técnicos e de marketing, programadores, usuários finais e clientes. Considerando esses diversos indivíduos, que possuem necessidades, interesses e preocupações distintas com relação ao produto a ser desenvolvido, podemos perceber que conflitos devam surgir devido às preferências divergentes entre os *stakeholders*, de modo que talvez seja impossível recomendar uma única configuração que satisfaça-os razoavelmente.

É perceptível a necessidade de uma ferramenta para dar suporte ao processo e aos

participantes. Um ambiente integrado que une a modelagem de *features* com a recomendação de configurações para os *stakeholders* envolvidos no processo seria capaz de solucionar o problema de maneira transparente para os usuários. Portanto, apresentamos neste trabalho o SACRES 2.0, um plug-in para a IDE Eclipse que implementa uma nova abordagem (STEIN; NUNES; CIRILO, 2014) utilizando um metamodelo para dar suporte à configuração de modelos de *features* com base nas preferências de múltiplos *stakeholders*. Cada configuração de *stakeholder* representa as preferências estabelecidas por um indivíduo sobre as *features* do modelo, e com base nessas configurações buscamos uma configuração final que satisfaça da melhor maneira os *stakeholders* envolvidos utilizando sete estratégias baseadas na teoria da escolha social.

1.1 Organização

O restante do trabalho está organizado com a seguinte divisão de capítulos:

- capítulo 2: contém a fundamentação teórica a respeito do trabalho, que foi realizada através de uma revisão bibliográfica principalmente sobre Linhas de Produto de Software, Modelos de *Features* e Recomendação de Configurações baseada em Preferências;
- capítulo 3: apresenta as tecnologias envolvidas no desenvolvimento do trabalho, chamadas de FeatureIDE e SACRES;
- capítulo 4: mostra com detalhes a implementação da ferramenta com a solução proposta, apresentando suas funcionalidades e arquitetura, sua interface e um exemplo com os resultados obtidos;
- capítulo 5: contém os testes de desempenho realizados;
- capítulo 6: apresenta a conclusão e possíveis trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Para melhor compreensão do trabalho desenvolvido, a seguir apresentamos a definição de alguns conceitos fundamentais sobre os assuntos tratados. O conteúdo deste capítulo foi dividido em três partes: (i) linhas de produto de software, (ii) modelo de *features* e (iii) recomendação de configurações baseada em preferências.

2.1 Linhas de Produto de Software

A motivação por trás de uma linha de produto é a possibilidade de reúso de componentes de uma mesma arquitetura. Essa ideia não é nova, pois pode ser observada sendo posta em prática na humanidade desde os tempos antigos, e como exemplo podemos citar a construção das pirâmides do Egito. Atualmente podemos compreender melhor a ideia através de um exemplo bem comum: uma linha de produção de carros. Quando vamos comprar um carro, podemos escolher o tipo de motor (1.0, 1.6, etc.), transmissão automática ou manual, a quantidade de portas, optar por ar-condicionado ou não, dentre várias outras opções. Neste caso, se pensarmos no conjunto de carros de um determinado modelo e de uma determinada marca de automóveis, podemos observar que eles constituem uma família de produtos, pois possuem algumas características comuns e algumas características variáveis.

A possibilidade de decisão do consumidor entre essas características variáveis caracteriza uma customização em massa, que proporciona uma produção em larga escala de produtos adaptados às necessidades individuais de cada consumidor. Quando combinamos essa customização em massa com desenvolvimento baseado em plataformas (uma plataforma é uma base tecnológica usada para a construção de outras tecnologias ou processos) alcançamos o reúso de uma base tecnológica comum. Assim podemos entregar ao consumidor um carro com as características que ele deseja, ou muito próximas delas de uma maneira mais fácil e mais barata do que projetar e construir desde o início um modelo de carro específico para o cliente.

Essas ideias foram utilizadas na área de desenvolvimento de software, e assim chegamos na criação de Linhas de Produto de Software (LPS). Segundo (VAN DER LINDEN; POHL, 2005) Linhas de Produto de Software referem-se a técnicas de engenharia para a criação de sistemas de software similares a partir de um conjunto compartilhado de partes do software, usando uma forma sistemática para a construção de aplicações. Algumas das abordagens que botam em prática essas técnicas serão detalhadas ao longo dessa seção. Seguindo a mesma referência, a engenharia de LPS é um paradigma para desenvolver aplicações de software (sistemas intensivos de software e produtos de software) usando plataformas e customização em massa, como já vimos anteriormente.

A definição de (CLEMENTS; NORTHROP, 2002) diz que uma LPS é um conjunto

de sistemas intensivos de software que compartilham um conjunto comum de *features* gerenciáveis, as quais satisfazem necessidades específicas de um segmento específico de mercado ou missão, que são desenvolvidos a partir de um conjunto de recursos comuns segundo um processo definido. De acordo com (CZARNECKI; EISENECKER, 2000), uma *feature* (neste contexto, usualmente traduzida como "característica") é uma propriedade do sistema que é relevante para algum *stakeholder* e que é usada para capturar pontos em comum ou para discriminar entre produtos em uma linha de produtos. Esse conceito e sua importância serão discutidos mais detalhadamente na próxima seção.

Complementando a definição dada anteriormente, um *stakeholder* é um indivíduo que tem um interesse no domínio, como por exemplo: gerentes técnicos e de marketing, programadores, usuários finais e clientes. Um domínio por sua vez é um corpo especializado de conhecimento, ou uma área de especialização, ou uma coleção de funcionalidades relacionadas. Na Seção 2.3 discutiremos a influência dos *stakeholders* no processo de configuração de produtos em uma LPS.

A Figura 2.1 ilustra a ideia básica do processo de desenvolvimento de uma LPS: a partir de uma base de recursos e através de um conjunto de decisões de produtos, passando pela fase de produção propriamente dita, temos como resultado um grupo de produtos especializados de acordo com as características escolhidas pelo(s) cliente(s).

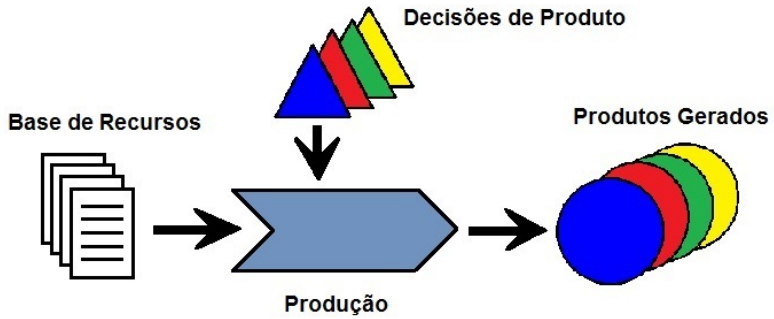


Figura 2.1: Processo de geração de produtos em uma LPS.

As principais vantagens do uso de LPS são: menor custo de desenvolvimento (é necessário um investimento inicial, porém seu retorno é obtido em uma média de três sistemas), melhora na qualidade do software (devido ao reúso) e redução de *time-to-market* (tempo entre a concepção de um produto e a sua disponibilização para venda, devido a menores ciclos de desenvolvimento). Dentre outras vantagens podemos citar: redução de esforço para manutenção, melhor estimação de custos e os benefícios aos consumidores (melhor qualidade e menor preço).

Entretanto, existem algumas desvantagens na utilização dessas técnicas: alto risco devido ao investimento inicial (que pode ser inútil se requerimentos importantes mudarem), maior *time-to-market* para o primeiro produto baseado na arquitetura da LPS e a necessidade de engenheiros especializados, junto com um gerenciamento técnico e organizacional.

Existem três atividades essenciais para o desenvolvimento de LPS (CLEMENTS; NORTHROP, 2002). Estas são detalhadas a seguir.

- Desenvolvimento de Recursos de Núcleo (*Core Asset Development*): Recursos de núcleo são os artefatos e recursos reusáveis que formam a base da LPS e podem incluir: a arquitetura, documentação, especificações, componentes reusáveis de software e casos de teste, entre outros. O objetivo desta atividade é o estabelecimento

de uma capacidade de produção de produtos. Também é chamado de engenharia de domínio.

- Desenvolvimento de Produto (*Product Development*): O objetivo dessa atividade, como seu nome sugere, é o desenvolvimento de um produto que atenda a um nicho de consumidores ou de mercado. Ela depende de três fatores: do escopo da linha de produção (descrição dos produtos que constituem a linha de produção ou que a linha de produção é capaz de incluir), da base de recursos de núcleo (todos recursos de núcleo que formam a base para a produção dos produtos) e do plano de produção (determina como os produtos são produzidos a partir dos recursos de núcleo). Também é chamado de engenharia de aplicação.

- Gerenciamento (*Management*): Seu objetivo principal é a distribuição de recursos, coordenação e supervisionamento das outras atividades, e é dividido em dois tipos. O gerenciamento organizacional identifica as restrições de produção e determina a estratégia de produção. O gerenciamento técnico (ou de projeto) por sua vez supervisiona as outras duas atividades para garantir que os responsáveis pela construção dos recursos de núcleo e dos produtos estão envolvidos nas atividades necessárias, seguem os processos definidos pela linha de produção e coletam dados suficientes para controlar o progresso.

Figura 2.2: As três atividades essenciais para o desenvolvimento de uma LPS.

Na Figura 2.2 retirada de (CLEMENTS; NORTHROP, 2002), cada círculo representa uma das atividades essenciais. Todas as três atividades são conectadas entre si e estão em movimento constante. Elas são altamente iterativas e podem ocorrer em qualquer ordem.

Existem três abordagens para o desenvolvimento de LPS (KRUEGER, 2002):

- Proativa: Considera todos os sistemas desenvolvidos anteriormente e começa a produção com o desenvolvimento dos recursos de núcleo.

- Reativa: A linha de produto aumenta incrementalmente devido à demanda por novos produtos ou novos requisitos para produtos existentes.

- Extrativa: Características comuns e variáveis são extraídas de sistemas existentes para formar uma nova LPS.

Essas abordagens podem ser atacadas incrementalmente e raramente são lineares. Além disso, elas não são necessariamente mutualmente exclusivas, como por exemplo, podemos iniciar utilizando a abordagem extrativa e depois mudarmos para a reativa, a fim de evoluir a linha de produto ao longo do tempo.

2.2 Modelo de Features

Para representarmos o conjunto de *features* disponíveis em uma LPS, é comum usarmos um modelo de *features* (ou modelo de características) (KANG et al., 1990). Seu objetivo é a modelagem das propriedades comuns e variáveis dos produtos possíveis de uma LPS, incluindo suas interdependências. Complementando a descrição dada na seção anterior, uma *feature* é uma característica do sistema que é visível para um *stakeholder*. Ela permite a expressão de partes comuns e variáveis entre as instâncias e representa requisitos reusáveis e configuráveis. Uma *feature* pode ser (CZARNECKI; EISENECKER, 2000):

- obrigatória: Deve estar presente em todos os produtos;
- opcional: Pode ou não estar presente em um produto;
- alternativa: Deve estar presente de forma exclusiva (analogamente à operação lógica *XOR*), ou seja, apenas uma entre um conjunto de *features* deve ser escolhida;
- relação de Ou: Deve estar presente de forma inclusiva (analogamente à operação lógica *OR*), isto é, pelo menos uma deve ser escolhida, mas podemos incluir até o conjunto todo.

Além disso, podemos definir restrições entre certas combinações de *features* existentes. Desse modo podemos garantir que uma determinada *feature* não poderá estar presente no produto se uma outra *feature* não estiver. Também podemos criar relações entre *features* de maneira que as tornem mutualmente excludentes. Esses são apenas dois exemplos do que podemos garantir com a criação de restrições.

Como uma *feature* pode ter *sub-features* (ou seja, ter *features* que sejam suas filhas), esse conjunto das *features* de um produto é organizado na forma de uma árvore e é chamado de modelo de *features*. A Figura 2.3 apresenta algumas possibilidades de relacionamento entre as *features* através de um exemplo abstrato contendo *features* com nomes não significativos, porém para um modelo de produtos concretos o funcionamento seria o mesmo, mudando apenas o nome dos nodos. Ela foi gerada utilizando um plug-in para a IDE (*Integrated Development Environment*) Eclipse chamado FeatureIDE[1], que será detalhado no capítulo seguinte, junto com outras tecnologias relacionadas.

No exemplo descrito na Figura 2.3 podemos observar uma árvore composta de nodos e arestas. O nodo mais acima (chamado de Exemplo), isto é, a raiz da árvore, representa um conceito. Ele é abstrato e não é uma *feature* propriamente dita, e normalmente serve de referência para a LPS representada no modelo. O restante dos nodos são concretos e representam todas as *features* contidas no modelo. A relação definida pelas arestas é intuitiva: se um nodo está abaixo de outro e ligado a ele através de uma aresta, significa

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

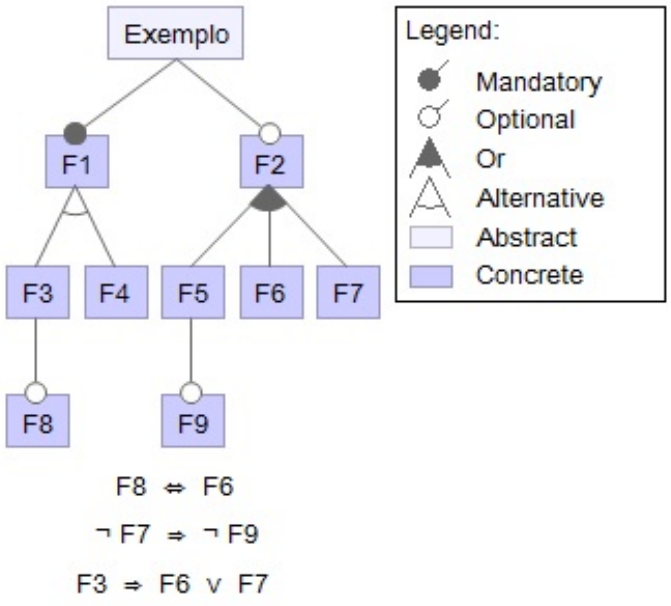


Figura 2.3: Exemplo de um Modelo de *Features*.

que ele é considerado filho do nodo que está acima. Assim sendo, para o nodo filho estar presente no produto, o nodo pai também deve estar presente. Além disso, um nodo pode ter mais de um filho, isto é, ser pai de um grupo de *features*.

Features obrigatórias são marcadas com um círculo preenchido em cima de seu nome (nodo F1), e as opcionais, por sua vez, são marcadas com um círculo vazio (nodos F2, F8 e F9). Um conjunto de *features* alternativas é representado por um arco não preenchido que conecta todas as *features* do grupo (nodos F3 e F4). De maneira semelhante, um conjunto de *features* ligadas entre si através de uma relação de ou é expresso com um arco conectando as *features* do grupo, porém diferentemente do grupo de *features* alternativas, este arco é preenchido (nodos F5, F6 e F7).

Abaixo da árvore estão listadas as restrições presentes no modelo. Foram usados alguns símbolos nestas notações: o símbolo de implicação ($\Rightarrow$), o operador se e somente se ($\Leftrightarrow$), o de ou lógico ($\vee$) e o negativo ($\neg$). Também é possível usar o operador de e lógico ($\wedge$), porém ele não está presente nesse exemplo. Através de algumas restrições podemos garantir várias relações entre as *features* presentes no modelo.

A primeira restrição nos diz que os nodos F8 e F6 devem ambos estar presentes, ou nenhum deles pode estar presente. É importante ressaltar que para o nodo F8 estar presente, o seu nodo pai (F3) também deve estar presente. Desse modo, caso selecionarmos o nodo F6, isso implicará na inclusão dos nodos F8 e F3, e além disso, na exclusão do nodo F4, pois ele possui uma relação de alternativa com o nodo F3. Devido a essas relações, os nodos F4 e F8 são conflitantes, pois apenas um deles pode estar presente nesse exemplo. A segunda restrição garante uma relação de dependência entre os nodos F7 e F9, de modo que o nodo F9 só pode estar presente se o nodo F7 também estiver. Entretanto, o contrário não é verdadeiro, pois podemos selecionar o nodo F7 e não selecionar o nodo F9. Uma forma de lermos essa restrição pode ser: a ausência do nodo F7 implica na ausência do nodo F9. A terceira e última restrição define que caso o nodo F3 seja selecionado, obrigatoriamente devemos selecionar o nodo F6, ou o nodo F7, ou ambos. Caso a operação entre os dois nodos fosse a de e lógico ($\wedge$), isso implicaria que necessariamente os dois

nodos deveriam estar presentes.

O objetivo desse pequeno exemplo foi mostrar como podemos representar o conjunto de *features* de uma LPS de uma maneira de fácil compreensão. Também mostramos algumas das relações que podemos garantir entre as *features* presente no modelo, seja por meio da escolha do tipo delas ou pela criação de restrições.

2.3 Recomendação de Configurações baseada em Preferências

Agora que já mostramos a organização e o funcionamento de um modelo de *features*, podemos falar sobre a configuração de produtos a partir de um modelo definido. Uma configuração de um modelo de *features* é a seleção de um conjunto de *features* contidas nesse modelo. Quando esta seleção de *features* respeita as restrições presentes no modelo ela é chamada de configuração válida, caso contrário é chamada de inválida. Os indivíduos responsáveis pelas configurações são denominados *stakeholders*, que podem ser clientes que financiam o produto, usuários, entre outros (como mencionado na Seção 2.1). Dado que modelos de *features* podem ter um grande número de *features* disponíveis para seleção e um número exponencial de configurações possíveis, escolher uma configuração adequada pode ser não trivial e requer suporte computacional (MENDONCA; COWAN, 2010). Isso nos sugere que a quantidade de configurações possíveis que podemos ter com modelos mais complexos pode tornar o processo de configuração muito custoso computacionalmente.

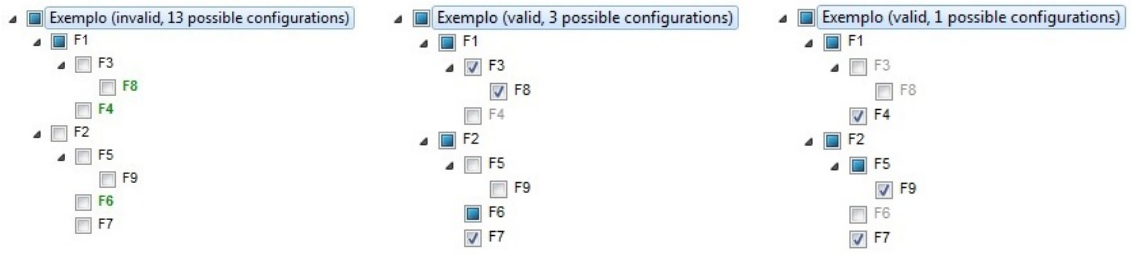
A ferramenta apresentada neste trabalho permite a especificação de configurações associadas a uma LPS. Entretanto, essas configurações não são as usualmente utilizadas, mas configurações onde novos tipos de preferências baseadas no metamodelo da nova abordagem (STEIN; NUNES; CIRILO, 2014) são especificadas. Além disso, recomendações são realizadas que resultam nessas configurações tradicionais.

O plug-in SPLConfig (MACHADO et al., 2014) também visa a recomendação automática de configurações ótimas maximizando a satisfação dos clientes e foi implementado de forma integrada com o FeatureIDE, porém não suporta a participação de múltiplos *stakeholders* no processo de configuração de um mesmo produto. Ele mantém o foco em outros requisitos e limitações, como custo, benefício do consumidor e orçamento, e utiliza um algoritmo baseado em busca para encontrar a melhor solução. Em outras palavras, procura ajudar os negócios da empresa através da busca de um conjunto de *features* que equilibra custo e satisfação de consumidor dado um determinado orçamento.

Neste trabalho usaremos o conceito da existência de múltiplos *stakeholders* envolvidos no processso de configuração de um mesmo produto. A ideia básica é um cenário onde temos um modelo de *features* e múltiplos *stakeholders* que expressam suas preferências através de configurações desse modelo, e o resultado esperado é uma configuração final que satisfaça da melhor maneira os *stakeholders* envolvidos. A participação de mais de um *stakeholder* nesse processo o torna mais complicado, pois alguns problemas podem surgir, como a ocorrência de preferências conflitantes entre eles, como por exemplo, quando a configuração de um *stakeholder* contém uma feature que um segundo *stakeholder* gostaria que não estivesse presente no produto.

Um simples exemplo de modelo mostrado na Figura 2.3 criado no FeatureIDE com nove *features* (excluindo a raiz da árvore) e com três restrições, possui 13 possibilidades de configuração válidas. Essa quantidade de configurações, por menor que possa parecer, pode gerar uma dificuldade na recomendação de uma configuração única que satisfaça razoavelmente todos os *stakeholders* envolvidos. Na Figura 2.4 podemos obervar algumas

configurações que foram geradas a partir desse modelo.



(a) (b) (c)

Figura 2.4: Algumas configurações possíveis do Exemplo dado na Figura 2.3.

A configuração da Figura 2.4(a) está em branco, isto é, não sofreu nenhum tipo de configuração (assinaladas através do símbolo ✓ ou de um quadrado azul preenchido) por parte de um *stakeholder*, e nela podemos observar que o FeatureIDE calculou que este modelo possui 13 configurações válidas. Entretanto essa configuração não é válida (notar a palavra *invalid* entre os parênteses), pois nenhuma *feature* dos grupos de alternativas e de relação de ou foi escolhida. As configurações da Figura 2.4(b) e da Figura 2.4(c) são válidas, porém teríamos problemas em recomendar uma única configuração que satisfaça esses dois *stakeholders*, porque elas possuem preferências conflitantes, por exemplo: o *stakeholder* da Figura 2.4(b) prefere a *feature* F3, enquanto o da a Figura 2.4(c) prefere a F4, e essas duas *features* não podem estar presentes na configuração ao mesmo tempo devido a relação de alternativa entre elas. Do modo como foram expressas essas preferências, qualquer configuração proposta para satisfazer ambos os *stakeholders* não seria suficiente, pois um deles teria que abrir mão de uma ou mais preferências que ele exigiu que fossem selecionadas.

2.3.1 Metamodelo Utilizado

Uma solução para o problema de configuração de modelos de *features* a partir de preferências de múltiplos *stakeholders* foi proposta como trabalho desenvolvido (STEIN; NUNES; CIRILO, 2014). O trabalho recém citado foi usado amplamente como base para a escrita deste capítulo, de modo que resolvemos omitir todas ocorrências de referências a ele ao longo do texto a seguir para facilitar a legibilidade. A implementação do protótipo desenvolvido no trabalho mencionado será discutida mais detalhadamente na Seção 3.2.

Existem duas maneiras principais de expressar preferências: qualitativa, definindo uma ordem de importância entre as preferências, e quantitativa, definindo um grau de preferência para as *features*. O metamodelo definido no trabalho citado possibilita que os *stakeholders* expressem suas preferências quantitativas através de um valor numérico que representa o quanto eles desejam que uma *feature* esteja ou não presente no produto.

Temos dois tipos de restrições definidas: *hard* e *soft*. Ambas podem ser tanto positivas quanto negativas. As restrições do tipo *hard* positivas indicam que o *stakeholder* deseja que uma determinada *feature* faça parte da configuração. Quando um *stakeholder* deseja que a *feature* não faça parte da configuração, ele expressa uma restrição *hard* negativa. Já as restrições do tipo *soft* são descritas como um número real no intervalo [-1,1]. Elas representam o grau de preferência do *stakeholder* sobre uma *feature*, ou seja, é indicado o quanto esse *stakeholder* gostaria que essa *feature* estivesse presente na configuração, no caso de uma restrição *soft* positiva, ou o quanto ele gostaria que não estivesse presente na configuração, no caso de uma restrição *soft* negativa. Quanto maior o valor escolhido,

ou seja, quanto mais próximo de 1, maior é a intenção de que a *feature* faça parte da configuração e quanto mais próximo de -1, maior é a intenção de que ela não faça parte da configuração. O valor zero indica indiferença.

Dessa forma podemos dizer que uma configuração de um *stakeholder* é composta por um conjunto de restrições *hard* positivas e negativas e por um conjunto de restrições *soft* positivas e negativas associados a um modelo de *features*. Em outras palavras, uma configuração de um *stakeholder* é uma tupla $sh\_cfg = \langle fm, HC^+, HC^-, sc \rangle$ onde fm é o modelo de *features* associado com a configuração do *stakeholder*, HC^+ é o conjunto de *features* que devem estar presentes na configuração, HC^- é o conjunto de *features* que não devem estar presentes e $sc : Feature \rightarrow$ [-1,1] é uma função que especifica o grau de preferência sobre as *features*, usando restrições do tipo *soft*. É válido ressaltar que as preferências definidas nessas configurações devem obedecer as restrições definidas no modelo de *features*.

Feature	SH1	SH2	NSH1	NSH2	Recomendação
F1	+	+	+	+	✓
F2	+	+	+	+	✓
F3	+		+		✓
F4		+		0.6	
F5		+	-0.5	0.75	✓
F6	+		0.4	-	
F7	+	+	0.75	+	✓
F8	+		0.25	-0.1	
F9		+		+	✓

Tabela 2.1: Recomendação de configuração após adaptação de preferências.

A Tabela 2.1 visa unir os exemplos mostrados até agora, através da apresentação das configurações dos dois *stakeholders* da Figura 2.4 como um grupo de *stakeholders* que expressaram preferências sobre o modelo da Figura 2.3. As colunas SH1 e SH2 se referem às preferências do *stakeholder* da Figura 2.4(b) e do *stakeholder* da a Figura 2.4(c), respectivamente. O símbolo + representa restrições do tipo *hard* positivas, enquanto o símbolo - representa restrições do tipo *hard* negativas. Foram atribuídas restrições *hard* às preferências definidas na Figura 2.4, porém é impossível encontrar uma configuração que satisfaça ambos *stakeholders* devido à existência de conflitos entre as preferências (as *features* F3 e F4 são mutuamente excludentes). Então as configurações foram refeitas (colunas NSH1 e NSH2) adaptando as preferências utilizando também o conceito de restrições do tipo *soft*, representadas por um número positivo ou negativo. Assim sendo, foi possível recomendar um conjunto de *features* (representadas por ✓ na última coluna) que constitui uma configuração válida.

2.3.2 Satisfação de Stakeholder

Agora que definimos o metamodelo utilizado, temos que pensar em como vamos identificar uma configuração que seja ótima levando em consideração as preferências selecionadas individualmente pelo grupo de *stakeholders* envolvidos. Para isso, devemos estabelecer o quanto uma configuração satisfaz um *stakeholder*. Primeiramente, vamos excluir as configurações válidas que não podem ser consideradas soluções ótimas para o

nosso problema. Fazemos isso a partir do uso das restrições do tipo *hard* descritas anteriormente. Uma configuração é excluída caso: tenha pelo menos uma *feature* associada a uma restrição *hard* positiva que não esteja no conjunto de *features* selecionadas e/ou tenha pelo menos uma *feature* associada a uma restrição *hard* negativa que esteja no conjunto de *features* selecionadas. Isso elimina conflitos entre configurações de *stakeholders* cujas preferências *hard* escolhidas impossibilitariam a satisfação de preferências *hard* de outro(s) *stakeholder(s)*, como no caso de um deles requerer uma determinada *feature* e o outro achar inaceitável sua presença na configuração.

Após o processo de exclusão de configurações, obtemos um conjunto cujos membros são todas configurações válidas, com exceção das configurações que para as preferências de pelo menos um *stakeholder* existam restrições *hard* insatisfeitas. Desse modo, qualquer configuração desse conjunto está apta a ser recomendada como configuração ótima. Denominamos esse conjunto como conjunto de consideração.

Para medirmos o nível de satisfação de um *stakeholder* em relação à uma configuração desse conjunto usaremos as restrições do tipo *soft*. Uma restrição *soft* pode ser satisfeita de duas maneiras: quando o grau de preferência for positivo e a *feature* associada a ela está presente no conjunto de *features* selecionadas ou quando o grau de preferência for negativo e a *feature* associada a ela não está presente no conjunto de *features* selecionadas. Detalhando melhor a definição, um conjunto de *features* $SC_{Sat}(cfg, sh\_cfg) \subseteq sh\_cfg[fm[Features]]$ é denominado de conjunto de restrições *soft* satisfeitas de um *stakeholder* quando para todo $f \in SC_{Sat}$, ou $f \in cfg[F]$ e $sc(f) > 0$, ou $f \notin cfg[F]$ e $sc(f) < 0$.

A partir desse novo conjunto de restrições *soft* satisfeitas podemos definir o que será chamado de satisfação de *stakeholder*: é uma função $sh\_sat : Cfg \times SH\_Cfg \rightarrow \mathbb{R}$ que indica o quanto uma determinada configuração $cfg \in Cfg$ é satisfeita por uma configuração de *stakeholder* $sh\_cfg \in SH\_Cfg$. A satisfação é a soma de todas restrições *soft* satisfeitas, como mostrado abaixo.

$$sh\_sat(sh\_cfg, cfg) = \sum_{f \in SC_{Sat}(sh\_cfg, cfg)} |sc(f)|$$

2.3.3 Estratégias de Escolha Social

Já sabemos o modo como cada *stakeholder* avalia uma configuração do conjunto de consideração. Essa informação é expressa através de um número entre zero e a quantidade de *features* existentes no modelo. Portanto, esse número pode ser interpretado de duas maneiras: como classificação para cada configuração (através da atribuição de pontos) e como ordenação entre as configurações. Apenas com esses dados não podemos garantir a recomendação de uma configuração que satisfaça suficientemente um grupo de *stakeholders*, então foram usadas sete estratégias de escolha social diferentes para resolver esse problema. Todas elas escolhem apenas uma solução dentre todas configurações contidas no conjunto de consideração, que será classificada como ótima. A seguir descrevemos brevemente cada estratégia.

- *Average (AVG)*: Se preocupa em encontrar a satisfação global máxima dos *stakeholders*. É calculada a média das satisfações individuais dos *stakeholders*, então a maior média entre elas é selecionada.

$$\max \sum_{sh\_cfg \in SH\_Cfg} \frac{sh\_sat(cfg, sh\_cfg)}{|SH\_Cfg|}$$

- *Multiplicative (MUL)*: Também se preocupa em encontrar a satisfação global máxima dos *stakeholders*, porém ela multiplica as satisfações individuais dos *stakeholders* e seleciona a maior multiplicação calculada.

$$\max \prod_{sh\_cfg \in SH\_Cfg} sh\_sat(cfg, sh\_cfg)$$

- *Most Pleasure (MP)*: Se preocupa em encontrar a melhor satisfação de *stakeholder* possível, então ela seleciona o máximo entre o máximo de satisfações individuais de *stakeholders*.

$$\max \max sh\_sat(cfg, sh\_cfg)$$

- *Least Misery (LM)*: Se preocupa em não deixar os *stakeholders* muito insatisfeitos, o que pode ocorrer com as três estratégias acima. Para isso, ela seleciona o máximo entre o mínimo de satisfações individuais de *stakeholders*.

$$\max \min sh\_sat(cfg, sh\_cfg)$$

- *Borda Count (BC)*: Usa a satisfação do *stakeholder* para ordenar as configurações. Assim temos uma classificação de configurações da satisfação mais alta (melhor) até a satisfação mais baixa (pior), e baseando-se nessa classificação cada configuração recebe um número de pontos. A pior configuração recebe 0 pontos, a segunda pior recebe 1 ponto e assim sucessivamente. Finalmente, os pontos de cada configuração individual de *stakeholder* são somados e o máximo é selecionado.

$$\max \sum_{sh\_cfg \in SH\_Cfg} rank(cfg, sh\_cfg)$$

- *Copeland Rule (CR)*: Também considera a relação de ordenação dada pela satisfação do *stakeholder*. Ela seleciona a configuração que tem a máxima diferença entre o quão frequente uma configuração ganha de outras configurações e o quão frequente ela perde. $Win(cfg)$ é o conjunto de outras possíveis configurações cfg' tais que cfg ganha de cfg' segundo a maioria de votos levando em conta todos os *stakeholders*, isto é, de acordo com a concordância da maioria dos *stakeholders* de que cfg ganha de cfg'. Uma cfg ganha de uma cfg' de acordo com uma configuração de *stakeholder* sh_cfg se $sh\_sat(sh\_cfg, cfg) > sh\_sat(sh\_cfg, cfg')$. $Lose(cfg)$, por sua vez, é o conjunto das outras configurações possíveis cfg' tal que cfg' ganha de cfg.

$$\max |Win(cfg)| - |Lose(cfg)|$$

- *Average Number of Preferences (ANP)*: Considera apenas a satisfação das restrições *soft*, ignorando os graus de preferência dos *stakeholders*. Seleciona a configuração com o número máximo de restrições *soft* satisfeitas.

$$\max \sum_{sh\_cfg \in SH\_Cfg} \frac{SC\_sat(cfg, sh\_cfg)}{|SH\_Cfg|}$$

No caso de empate entre configurações dentro de uma mesma estratégia, nos levando a mais de uma solução ótima, é escolhida a configuração com o menor número de *features* selecionadas. Os *stakeholders* podem ser indiferentes a certas *features*, então quando não existe nenhuma preferência sobre uma *feature*, isso indica que provavelmente ela não é necessária. Se persistir o empate, uma configuração é escolhida aleatoriamente. Também é válido comentar que cada estratégia funciona de uma maneira diferente das outras, porém uma mesma configuração ótima pode ser selecionada por mais de uma estratégia.

Agora que apresentamos os fundamentos teóricos necessários para o entendimento do trabalho desenvolvido, seguimos com o próximo capítulo, que tem como objetivo a discussão das tecnologias (e das escolhas feitas sobre elas) que tornaram possível a realização deste trabalho.

3 TECNOLOGIAS

Neste capítulo apresentaremos as tecnologias envolvidas no desenvolvimento do trabalho e nos aprofundaremos em alguns pontos importantes que servirão para facilitar o entendimento da ferramenta desenvolvida.

Descrevendo sucintamente: a ferramenta desenvolvida é um plug-in para o Eclipse. Ela foi desenvolvida usando a linguagem de programação Java, uma linguagem orientada a objetos muito utilizada atualmente. Também foi usada a IDE Eclipse, que é um ambiente de desenvolvimento integrado amplamente utilizado para a criação de aplicações em diversas linguagens, principalmente em Java. O Eclipse permite que desenvolvedores contribuam com suas próprias funcionalidades à IDE na forma de plug-ins, e essa possibilidade de extensão é um dos motivos de seu alto uso entre eles.

Plug-ins são componentes de software que adicionam uma funcionalidade a uma aplicação existente. Através deles podemos aprimorar aplicações de várias maneiras diferentes, como por exemplo permitir novos formatos de arquivos para um reprodutor de vídeo e acrescentar novas funções a navegadores web. Existem muitos plug-ins disponíveis para o Eclipse, os quais proporcionam várias melhorias e novas funcionalidades à IDE, desde pequenas alterações na interface até grandes aplicações com os mais diversos propósitos.

Dentro do contexto deste trabalho podemos citar algumas ferramentas existentes com funcionalidades relacionadas à nossa abordagem. De caráter comercial, as duas principais são pure::variants[1] e Gears[2]. Ambas oferecem as possibilidades de criação de modelos de *features* com notações gráficas e de criação de configurações válidas, porém necessitam da compra de uma licença para seu uso. De caráter acadêmico, temos o SPLOT[3] (Software Product Lines Online Tools), que possibilita a criação, edição e configuração de modelos de *features* na forma de árvore, entre outras funcionalidades. Esse projeto é apresentado na forma de um *website* com centenas de modelos disponíveis para os usuários, entretanto ele não fornece a possibilidade de integração com código.

Ainda de caráter acadêmico, ressaltamos três ferramentas que foram desenvolvidas como plug-ins para o Eclipse: XFeature[4], FMP[5] e FeatureIDE[6]. O XFeature suporta a modelagem de famílias de produtos e permite que seus usuários definam seus próprios metamodelos de *features*, porém não é possível realizar configurações de produtos. O FMP possui características similares ao XFeature, incluindo a ausência da possibilidade de criação de configurações. Já o FeatureIDE dispõe das mesmas funcionalidades (além

[1] http://www.pure-systems.com/Variant_Management.49.0.html
[2] http://www.biglever.com/solution/product.html
[3] http://www.splot-research.org/
[4] http://www.pnp-software.com/XFeature/
[5] http://gsd.uwaterloo.ca/fmp
[6] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

de outras adicionais) e diferentemente dos demais, ele permite criar e editar configurações de produtos. Dentre as opções citadas escolhemos o FeatureIDE para servir como base da nossa ferramenta, por se enquadrar melhor com os nossos objetivos e ter os requisitos necessários para possibilitar o desenvolvimento do nosso plug-in, além de possuir uma documentação detalhando possíveis pontos de extensão. A seguir mostramos algumas das características do FeatureIDE, seu funcionamento e alguns detalhes do plug-in.

3.1 FeatureIDE

O FeatureIDE (THüM et al., 2014) é um framework baseado no Eclipse que suporta FOSD (Feature-Oriented Software Development(APEL; KÄSTNER, 2009)). Seu foco principal é a cobertura de todo o processo de desenvolvimento e a incorporação de ferramentas para a implementação de LPS em um ambiente de desenvolvimento integrado (IDE). O objetivo do FeatureIDE é a redução de esforços para a construção de ferramentas para novas e existentes técnicas de implementações de LPS providenciando análise de domínio, implementação de domínio, análise de requisitos e geração de software em um framework extensível.

FOSD pode ser usado para planejar e implementar LPS (processo chamado de Desenvolvimento de Recursos de Núcleo ou engenharia de domínio na Seção 2.1) e também para selecionar *features* e gerar programas customizados (processo chamado de Desenvolvimento de Produto ou engenharia de aplicação na Seção 2.1). Os autores dividiram o processo FOSD em quatro fases, as quais foram citadas no parágrafo anterior:

- Análise de domínio (*Domain analysis*): O objetivo é a captura das variabilidades e semelhanças de um domínio de sistema de software, que resulta em um modelo de *features*.

- Implementação de domínio (*Domain implementation*): Implementação de todos sistemas de software do domínio ao mesmo tempo, enquanto os recursos de núcleo são mapeados para *features*.

- Análise de requisitos (*Requirements analysis*): Requisitos são mapeados para as *features* do domínio, e as *features* necessárias para um sistema de software customizado são selecionadas, resultando em uma configuração.

- Geração de software (*Software generation*): Um sistema de software é construído automaticamente, dadas a configuração e a implementação do domínio.

O FeatureIDE está em constante desenvolvimento, e algumas de suas características e funcionalidades implementadas são: (i) total integração com o Eclipse, (ii) editor de modelo de *features* gráfico e baseado em texto (como visto no modelo gráfico construído na Figura 2.3), (iii) editor de restrições com assistente de conteúdo e checagem de semântica e sintaxe, (iv) editor de configurações com suporte a derivação de configuração válida (mostrado parcialmente na Figura 2.4), (v) uma seção exibindo estatísticas sobre a LPS, além de (vi) integração com diversas linguagens e ferramentas já existentes. A Figura 3.1 mostra uma captura de tela do Eclipse exibindo a perspectiva do FeatureIDE, nela podemos notar algumas de suas principais funcionalidades mencionadas neste parágrafo.

Podemos ressaltar alguns outros pontos positivos, a facilidade de uso sendo um dos seus principais. Primeiramente, o plug-in pode ser baixado e instalado de maneira simples

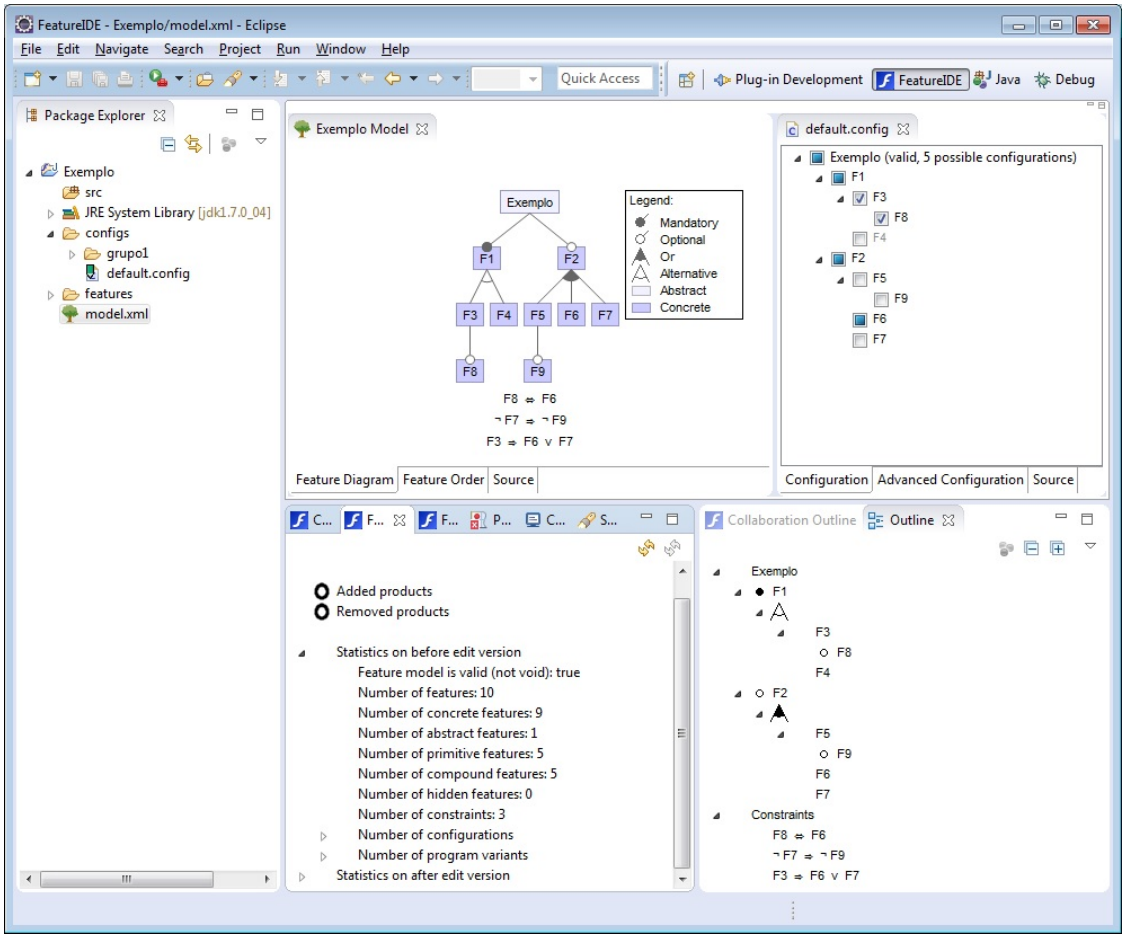


Figura 3.1: Perspectiva do FeatureIDE.

diretamente através do Eclipse. O *website* do FeatureIDE dispõe de tutoriais e documentação na forma de slides, vídeo de demonstração, capturas de tela, entre outros, com o propósito de ajudar seus usuários desde a instalação do plug-in e criação de um pequeno exemplo até a sua possível extensão por outros desenvolvedores. Essa documentação inclui uma descrição de como é possível adicionar funcionalidades ao plug-in através de pontos de extensão, indicando quais pontos devem ser estendidos para alcançar as alterações desejadas.

Esta seção não entrou em detalhes muito específicos, principalmente na parte de implementação do FeatureIDE, pois seu propósito é apenas dar uma visão geral do plug-in. No próximo capítulo nos aprofundaremos um pouco mais nos detalhes de algumas de suas funcionalidades e mostraremos como usá-lo juntamente com a nossa implementação. A seção a seguir mostra, também superficialmente, o SACRES, que possui uma implementação da solução proposta por (STEIN; NUNES; CIRILO, 2014). Ele foi desenvolvido como um protótipo para viabilizar um experimento com o objetivo de avaliar diferentes estratégias da teoria da escolha social.

3.2 SACRES

SACRES (STEIN; NUNES; CIRILO, 2014) é uma ferramenta independente desenvolvida em Java baseada no metamodelo da Seção 2.3.1, no conceito de satisfação de *stakeholder* da Seção 2.3.2 e nas estratégias de escolha social da Seção 2.3.3 que tem como

finalidade a recomendação de configurações para grupos de stakeholders. Diferentemente do FeatureIDE, o SACRES não permite a criação e edição de modelos de *features*, então ele usa uma biblioteca de modelos de *features* que oferece persistência para um banco de dados, e os modelos propriamente ditos estão presentes nesse banco. O restante das informações como grupos de *stakeholders* e configurações de *stakeholders* também são armazenadas no banco de dados.

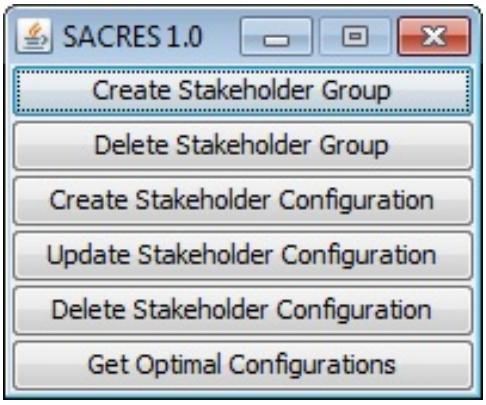
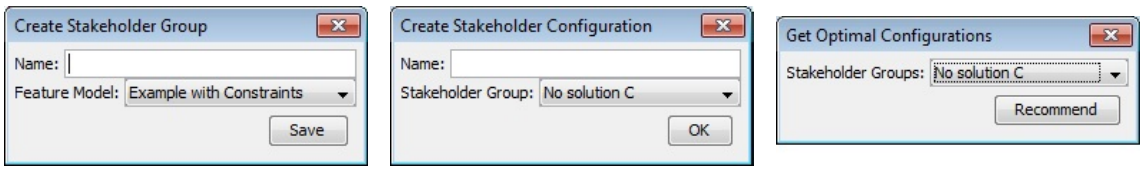


Figura 3.2: Tela inicial do SACRES.

A Figura 3.2 mostra a tela de interface inicial do SACRES, e nela podemos observar suas três principais funcionalidades:

- Criação de Grupo de *Stakeholder* (*Create Stakeholder Group*): Permite a criação de grupos de *stakeholders*. O usuário escolhe o nome do grupo e associa um modelo de *features* a esse grupo, como mostrado na Figura 3.3(a).
- Criação de Configuração de *Stakeholder* (*Create Stakeholder Configuration*): Permite a criação de configurações de *stakeholders*. O usuário escolhe o nome da configuração e associa um grupo de *stakeholders* (que por sua vez está associado a um modelo de *features*) à essa configuração, como mostrado na Figura 3.3(b).
- Obter Configurações Ótimas (*Get Optimal Configurations*): O usuário seleciona um grupo de *stakeholders* conforme a Figura 3.3(c) e o resultado da recomendação de configurações é exibido.



(a) Create Stakeholder Group (b) Create Stakeholder Configuration (c) Get Optimal Configurations

Figura 3.3: Telas das funções do SACRES.

Quando o usuário está selecionando suas preferências na configuração o modelo de *features* é exibido de uma maneira semelhante à do mesmo processo no FeatureIDE (Figura 2.4), e quando uma *feature* é selecionada é exibida ao usuário uma janela com três

opções: definir como restrição *hard* positiva, definir como restrição *hard* negativa e definir como restrição *soft* escolhendo um valor numérico entre -1 e 1. As restrições positivas ficam assinaladas com a cor verde, enquanto que as restrições negativas ficam assinaladas com a cor vermelha. Já as restrições do tipo *soft* são acompanhadas com o valor numérico ao lado do nome da *feature*. *Features* sem preferências definidas são escritas em preto. A Figura 3.4 exemplifica a interface da etapa de criação/edição de configuração detalhada nesse parágrafo.

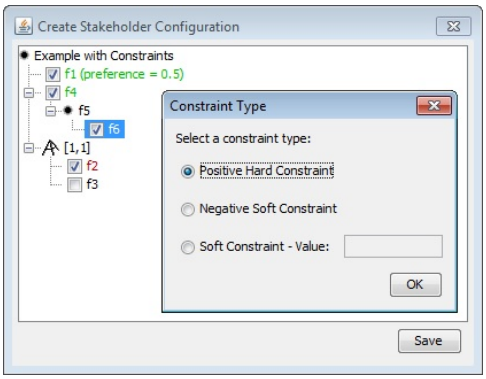


Figura 3.4: Tela de seleção de preferências do SACRES.

Caso as configurações de um determinado grupo de *stakeholders* resultem em uma recomendação de configurações válidas (configurações válidas envolvendo restrições dos tipos *hard* e *soft* foram discutidas na Seção 2.3.2) uma nova tela com as recomendações é exibida. A Figura 3.5 mostra a tela de resultados do SACRES, e nela podemos encontrar diversas informações relevantes: as configurações definidas pelo grupo de *stakeholders* (abas SH1 C, SH2 C e SH3 C), as configurações recomendadas para as sete estratégias de escolha social (restantante das abas) e uma pequena seção com algumas outras informações. Na parte debaixo da tela, onde está escrito *SATISFACTION*, são exibidos os seguintes dados: valor total das satisfações dos *stakeholders*, quantidade de configurações envolvidas, valor mínimo de satisfação de *stakeholder*, valor máximo de satisfação de *stakeholder*, média dos valores de satisfação de *stakeholder* e desvio padrão dos valores de satisfação de *stakeholder*. Mais abaixo na seção *PREFERENCES* os dados exibidos são análogos aos da seção *SATISFACTION*, porém desta vez levam em consideração a quantidade de restrições do tipo *soft* em vez do valor de satisfação de *stakeholder* que elas representam.

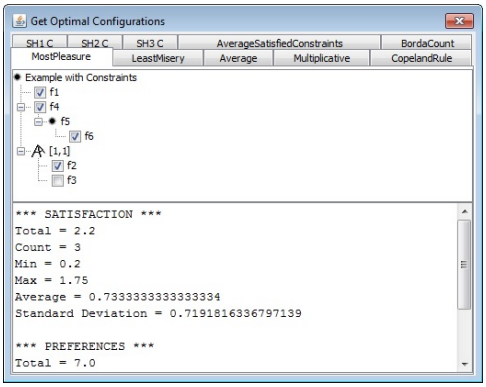


Figura 3.5: Tela de resultados do SACRES.

Nesta seção apresentamos a ferramenta chamada SACRES, mostrando suas funcionalidades e sua interface. Como a ferramenta que desenvolvemos se trata de uma integração do plug-in FeatureIDE com o trabalho realizado no SACRES esse capítulo teve o propósito de dar uma ideia geral sobre esses dois tópicos, mostrando o que cada um deles pode nos proporcionar e também o que não proporcionam, pois desse modo a união dos dois se mostra útil a fim de conseguirmos realizar todas etapas envolvidas na recomendação de configurações em uma única ferramenta. No capítulo a seguir nos aprofundaremos na parte de implementação da ferramenta e em alguns detalhes técnicos.

4 SACRES 2.0: PROJETO E IMPLEMENTAÇÃO

Nossa ferramenta realizou uma integração dos trabalhos descritos anteriormente chamados FeatureIDE e SACRES, unindo as suas funcionalidades com o objetivo de criar um plug-in que consiga cobrir todo o processo de recomendação de configurações desde a criação do modelo de *features* até a exibição dos resultados das recomendações das estratégias usadas. Decidimos chamá-la de SACRES 2.0 por se tratar de uma evolução do protótipo original. Assim sendo, o FeatureIDE e a nossa ferramenta dividem os trabalhos entre si, ou seja, cada um é responsável por um conjunto de etapas no processo. Ao longo deste capítulo mostraremos as funções exercidas individualmente por eles, juntamente com detalhes da nossa implementação, incluindo elementos da interface, os arquivos envolvidos na execução do plug-in e resultados obtidos.

4.1 Criação de Modelos de *Features* com o FeatureIDE

O FeatureIDE é responsável principalmente pelas primeiras etapas do processo. Podemos utilizar um projeto previamente criado com o FeatureIDE contendo um modelo de *features* associado a ele ou partir do zero e seguir os passos descritos abaixo.

1. **Criação de um projeto do FeatureIDE.** Para utilizar a ferramenta é necessário um projeto no Eclipse no formato padrão do FeatureIDE. Ele contém um arquivo no formato XML que é usado para armazenar o modelo de *features* associado ao projeto.

2. **Criação das *features*.** Após a criação do projeto deve-se criar o modelo de *features*. Nessa etapa o usuário cria os nodos da árvore e seleciona suas características, incluindo tipos das *features*, relação entre elas e possivelmente realiza a criação de grupos de *features*.

3. **Criação das restrições.** Também podem ser criadas restrições entre as *features* do modelo. O FeatureIDE provê um editor para facilitar essa etapa ao usuário que é capaz de checar a sintaxe e a semântica das expressões lógicas que representam as restrições.

Após a execução destes passos temos como resultado um projeto do FeatureIDE contendo um modelo de *features*. Esse modelo é armazenado em um arquivo no formato XML que contém as informações sobre todas as *features* presentes no modelo e também sobre suas restrições, caso existam. É possível observar seu conteúdo textual através da mesma *view* que exibe a árvore do modelo graficamente, entretanto não é aconselhável fazer alterações diretamente nele, pois isso pode tornar o modelo inválido. O usuário

pode fazer suas edições através da interface gráfica disponibilizada pelo FeatureIDE caso deseje alterar algo após a criação do modelo, e assim as mudanças serão refletidas no arquivo XML.

Dentro de qualquer projeto do FeatureIDE existe uma pasta chamada por padrão de "configs" que tem o objetivo de armazenar configurações. Um arquivo com o nome de "default.config" é criado nessa pasta e ele representa uma configuração para o modelo de *features* associado ao projeto. Podemos visualizar e editar o conteúdo desse tipo de arquivo através de uma *view* própria do FeatureIDE que exibe as *features* do modelo e também quadrados que indicam a sua presença ou ausência na configuração (ver o exemplo na Figura 2.4, mostrada no Capítulo 2). Iremos utilizar essa funcionalidade para a visualização das configurações recomendadas pelo nosso plug-in ao final do processo. Analogamente ao caso do arquivo do modelo de *features* podemos observar o conteúdo textual desse arquivo na *view* usada para a sua exibição gráfica, mas não é aconselhável a alteração diretamente por ali. O arquivo de configuração que o FeatureIDE utiliza é simplesmente uma lista de todas as *features* presentes na configuração distribuídas na forma de uma *feature* por linha.

4.2 Funcionalidades e Arquitetura

Agora que já descrevemos algumas das funcionalidades oferecidas pelo FeatureIDE que são utilizadas pela nossa ferramenta, iremos discutir a arquitetura do SACRES 2.0, o algoritmo de execução da ferramenta e as funcionalidades implementadas.

A Figura 4.1 mostra a arquitetura do plug-in desenvolvido com a interação entre as partes envolvidas. Nela, podemos ver que o FeatureIDE é o responsável pela criação e edição do modelo de *features* e pela visualização das configurações recomendadas, como foi mostrado na seção anterior. O SACRES 2.0 contém as classes com suporte ao metamodelo e a implementação dos algoritmos de recomendação para as estratégias de escolha social usadas. Além disso, o plug-in também é responsável pelo gerenciamento dos arquivos, desde os novos arquivos com as configurações de *stakeholder* baseadas no metamodelo até os resultados das recomendações que podem ser visualizados no FeatureIDE. O SACRES 2.0 faz o uso do CSP Solver CHOCO[1] de forma a eliminar de maneira mais eficiente as configurações consideradas inválidas pelas restrições do tipo *hard*, gerando o chamado conjunto de consideração, utilizado pelas estratégias de escolha social para buscar as configurações ótimas.

O processo de recomendação de configurações de produtos realizado pelo SACRES 2.0 pode ser descrito em sete passos, considerando que nenhum grupo contendo configurações de *stakeholders* foi previamente criado. Caso já existam grupos e configurações no projeto podemos pular os passos 3 e 5. Algumas destas funcionalidades serão melhor detalhadas juntamente com sua interface na próxima seção. Os passos para o processo de recomendação são descritos a seguir.

1. **Leitura do modelo de *features*.** O arquivo XML do FeatureIDE que representa o modelo de *features* atual é aberto e seu conteúdo é lido. Isto é realizado através de um clique em cima da *view* que exibe o modelo.

2. **Construção do modelo de *features*.** Após a leitura do arquivo XML o modelo é construído e armazenado durante a execução da ferramenta para ser utilizado nas partes de criação e recomendação de configurações.

[1] `http://choco-solver.org/?q=Choco3`

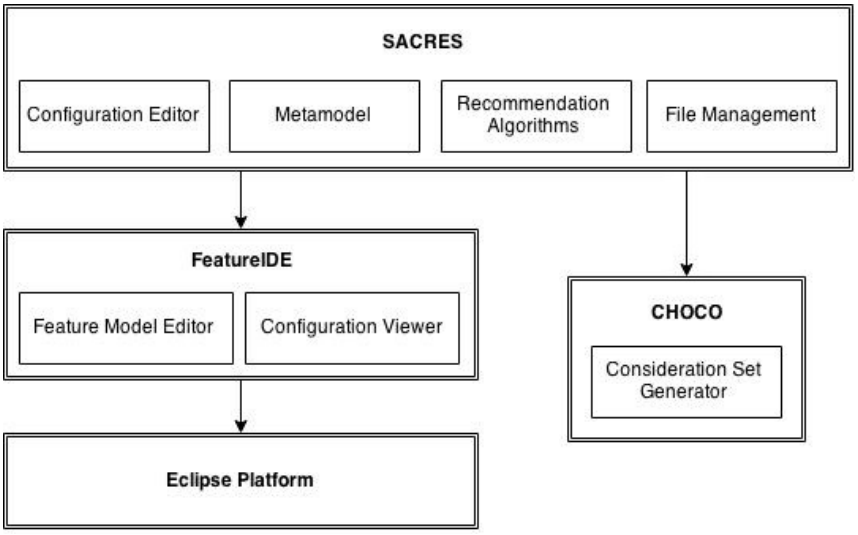


Figura 4.1: Arquitetura da ferramenta.

3. **Criação de grupo de *stakeholders*.** Grupos podem ser criados através de novas pastas localizadas dentro da pasta "configs" do projeto. Cada subpasta representa um grupo de *stakeholders*.

4. **Seleção de grupo de *stakeholders*.** Um grupo deve ser selecionado na *view* da ferramenta para ser possível a criação de configurações para esse grupo ou a recomendação de configurações para ele.

5. **Criação de configuração de *stakeholder*.** Após a seleção de um grupo é possível a criação de configurações baseadas no metamodelo utilizado pelo SACRES. Também podemos visualizar o conteúdo de configurações criadas previamente. Os arquivos são salvos dentro da pasta de seu grupo na forma de uma preferência por linha, que contém o nome da *feature*, tipo da restrição e seu valor, todos separados por vírgula.

6. **Recomendação de configurações ótimas.** Quando um grupo contendo mais de uma configuração de *stakeholder* é selecionado, podemos executar o algoritmo de recomendação de configurações ótimas, que por sua vez é dividido em quatro etapas.

   (a) **Leitura das configurações.** Os arquivos contendo as configurações dos *stakeholders* do grupo baseadas no metamodelo são carregadas.

   (b) **Geração de configurações válidas.** O CHOCO é o responsável pela geração de configurações válidas, resultando no conjunto de consideração. As restrições do tipo *hard* são analisadas para excluir configurações inválidas. Caso nenhuma configuração válida exista a execução é interrompida nesta etapa e um erro é exibido.

   (c) **Geração das configurações ótimas.** É feita a seleção da configuração ótima dentre as soluções viáveis de acordo com o critério de cada estratégia.

   (d) **Geração dos arquivos de resultados.** As configurações ótimas para cada estratégia são traduzidas para arquivos de configuração seguindo o padrão do

FeatureIDE. Também é criado um arquivo texto com algumas informações sobre a satisfação dos *stakeholders* e número de preferências satisfeitas.

7. **Visualização dos resultados.** As configurações podem ser visualizadas através da *view* específica do FeatureIDE. Além disso o arquivo texto de resultados pode ser aberto pelo editor de texto do Eclipse.

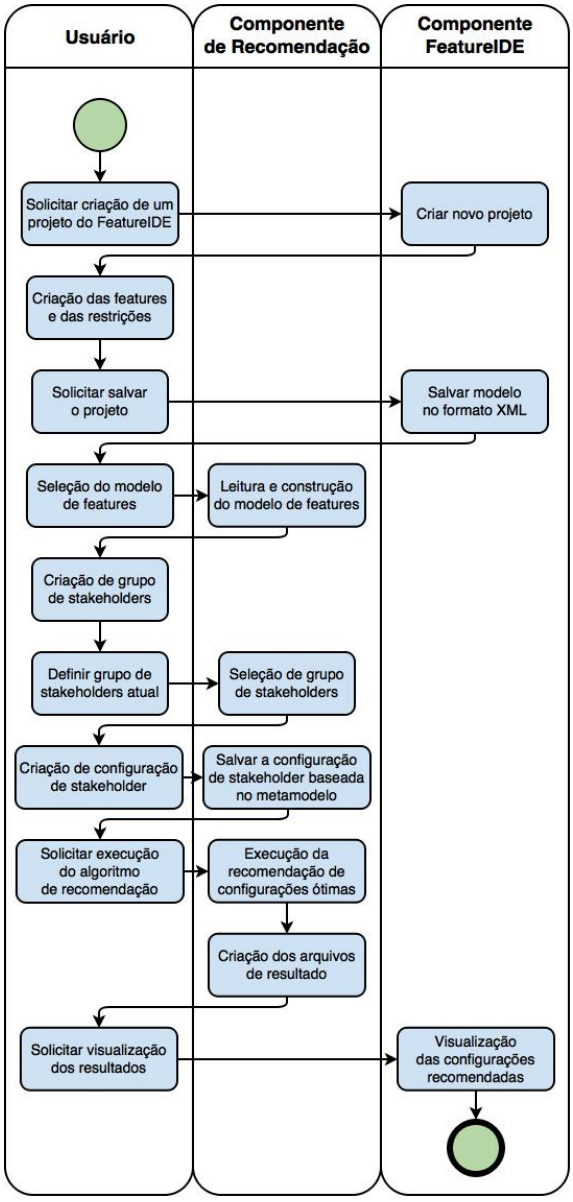


Figura 4.2: Ilustração do processo.

A Figura 4.2 ilustra o processo descrito nos passos acima (incluindo os três necessários para a criação do modelo de *features* com o FeatureIDE mostrados na Seção 4.1) indicando quem é o responsável por cada um deles. A maioria das ações do usuário refletem na execução de alguma tarefa por parte dos componentes integrados.

Como apresentado anteriormente foram utilizadas sete estratégias baseadas na teoria da escolha social para a recomendação das configurações. Cada uma delas gera uma configuração de produto, de acordo com suas características. Sendo assim, após a execução

do algoritmo da ferramenta, teremos até sete configurações distintas como resultado, visto que estratégias diferentes podem gerar a mesma configuração.

O plug-in precisa do acesso ao modelo de *features* do FeatureIDE através da interface gráfica para realizar o passo número 1 descrito acima. Para isso foi adicionada uma dependência ao projeto, incluindo o pacote "de.ovgu.featureide.fm.ui" pertencente ao FeatureIDE. Outras dependências presentes no projeto são "org.eclipse.ui" e "org.eclipse.core.resources", que servem para possibilitar a criação da nova *view* da ferramenta e o gerenciamento dos arquivos utilizados por ela.

4.3 Interface

Nesta seção apresentaremos a interface com o usuário oferecida pelo SACRES 2.0. A interação com o plug-in acontece principalmente através de uma *view* que conta com grande parte das funcionalidades implementadas. A *view* padrão do Eclipse utilizada para exibir os arquivos do projeto também faz parte do funcionamento da ferramenta, visto que os grupos de *stakeholders* são gerenciados diretamente nas pastas do projeto. A interface oferecida pelo FeatureIDE também é utilizada no processo, pois é através dela que criamos o modelo de *features* e visualizamos as configurações recomendadas.

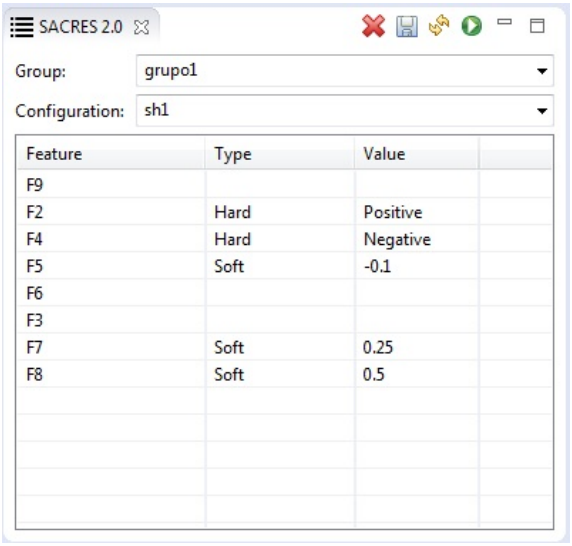


Figura 4.3: *View* do SACRES 2.0.

A Figura 4.3 mostra a *view* da ferramenta durante sua execução com o modelo da Figura 2.3 associado ao projeto atual e exibindo uma configuração salva. A *view* pode ser dividida em três partes: (i) a seção que seleciona as pastas e arquivos; (ii) a tabela com as *features* para alteração de configurações de *stakeholder*; e (iii) os botões de funcionalidade.

Grupos de *stakeholders* são representados por pastas dentro de um projeto do FeatureIDE e configurações de *stakeholders* são representadas por arquivos dentro dessas pastas. Na *view* primeiramente deve-se selecionar um grupo pertencente à lista de grupos associados ao modelo de *features*. Depois é possível abrir configurações salvas dentro desse grupo ou criar uma nova configuração.

A tabela de configuração é dividida em três colunas, que listam o nome das *features*, o tipo da restrição associada a ela e o valor dessa restrição. O modelo de exemplo possui dez nodos na árvore, porém apenas oito estão sendo listados na tabela, pois o nodo raiz

é abstrato não sendo uma *feature* propriamente dita e o nodo F1 representa uma *feature* obrigatória, assim sendo não é possível expressar preferências sobre ele. Ao estabelecer preferências, primeiramente o usuário seleciona o tipo (*hard* ou *soft*), e depois escolhe seu valor. O tipo é selecionado através de uma *combobox*. Caso a restrição seja do tipo *hard*, o usuário escolhe o valor da mesma maneira que o tipo foi escolhido, ou seja, através de uma *combobox*, e caso o tipo seja *soft*, o usuário digita o valor numérico na célula correspondente.

No canto superior direito da *view* encontram-se quatro botões com as seguintes funções.

- Reset Current Configuration: Apaga a configuração atual. Todos os campos da tabela são limpos e o usuário pode começar do zero a criação de uma nova configuração.
- Save Configuration: Salva a configuração atual. Ao salvar uma configuração, o usuário deve escolher um nome para ela, digitando na caixa “Configuration". As configurações são salvas com a extensão “.cfg”.
- Refresh: Atualiza as informações da *view*, incluindo o conteúdo das *comboboxes* de grupos e configurações.
- Get Optimal Configurations: Executa os algoritmos de recomendação de configurações. Quando executamos os algoritmos de recomendação uma pasta com os resultados é criada, e caso não existam configurações válidas para serem recomendadas uma mensagem de erro é exibida.

Após a execução das recomendações os arquivos de resultado são gerados, como podemos observar na Figura 4.4. Uma pasta chamada “recommendations” é criada dentro da pasta do grupo. Ela contém sete arquivos que representam as configurações recomendadas pelas estratégias de escolha social no formato padrão do FeatureIDE “.config”. O nome de cada arquivo é referente à estratégia usada para a sua geração, e como foi dito anteriormente, podemos visualizar essas configurações resultantes através da *view* própria do FeatureIDE.

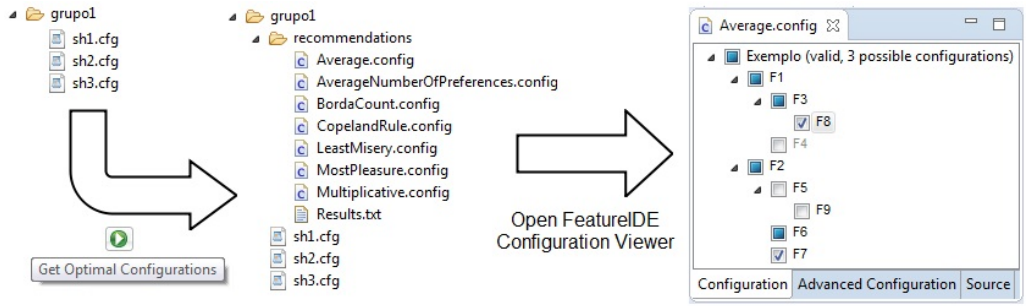


Figura 4.4: Arquivos resultantes.

Além dos arquivos de configuração presentes na pasta criada após a execução do algoritmo, também é criado um arquivo texto chamado “Results.txt”. Nele podemos coletar diversas informações a respeito das recomendações geradas. Na seção a seguir discutiremos melhor os resultados, incluindo o conteúdo deste arquivo texto, através de um exemplo simples.

4.4 Resultados

Para exemplificar os resultados de uma execução do algoritmo da ferramenta criamos um grupo contendo três *stakeholders*. O modelo de *features* ao qual o grupo foi associado foi exibido na Figura 2.3. A Tabela 4.1 mostra as preferências dos três *stakeholders*. Lembrando que os símbolos + e - representam restrições do tipo *hard* positivas e negativas, respectivamente, e que os valores numéricos representam restrições do tipo *soft*.

Feature	SH1	SH2	SH3
F2	+	0.6	+
F3			0.3
F4	-		
F5	-0.1		
F6			-0.15
F7	0.25	+	0.1
F8	0.5	0.2	
F9			

Tabela 4.1: Configurações do exemplo.

Como mencionado na seção anterior, após a execução da recomendação de configurações um arquivo texto chamado "Results.txt" é criado. A primeira linha dele contém o nome do grupo responsável pela sua geração e o restante é dividido em sete blocos, um para cada estratégia. Cada bloco começa com o nome da estratégia, e logo abaixo temos uma sequência de informações a respeito da satisfação dos *stakeholders* do grupo.

Satisfação	AVG	ANP	BC	CR	LM	MP	MUL
Total	2.2	2.2	2.1	2.2	2.1	2.05	2.2
Mínima	0.55	0.55	0.55	0.55	0.55	0.4	0.55
Máxima	0.85	0.85	0.8	0.85	0.8	0.85	0.85
Média	0.73	0.73	0.7	0.73	0.7	0.68	0.73
Desvio padrão	0.13	0.13	0.11	0.13	0.11	0.2	0.13

Tabela 4.2: Resultados referentes à satisfação de *stakeholder*.

A Tabela 4.2 apresenta o conteúdo do arquivo de resultados referente à satisfação dos *stakeholders*. Relembrando que as abreviações na primeira linha da tabela correspondem aos nomes das estratégias, *Average, Average Number of Preferences, Borda Count, Copeland Rule, Least Misery, Most Pleasure e Multiplicative*, respectivamente. A segunda linha corresponde ao total das satisfações, ou seja, a soma da satisfação individual de cada um deles. A terceira e quarta linhas correspondem à satisfação individual mínima e à satisfação individual máxima alcançadas, respectivamente. A quinta corresponde à média aritmética das satisfações e a sexta linha ao desvio padrão.

Após os dados no arquivo sobre satisfação de *stakeholder*, se encontram as informações referentes ao número de preferências satisfeitas, ou seja, à quantidade de restrições do tipo *soft* atendidas. Os tipos de informação são análogas às da parte de satisfação de *stakeholder*, como podemos observar na Tabela 4.3. Ela contém o total de preferências

Preferências	AVG	ANP	BC	CR	LM	MP	MUL
Total	8	8	7	8	7	7	8
Mínima	2	2	2	2	2	2	2
Máxima	3	3	3	3	3	3	3
Média	2.6	2.6	2.3	2.6	2.3	2.3	2.6
Desvio padrão	0.47	0.47	0.47	0.47	0.47	0.47	0.47

Tabela 4.3: Resultados referentes ao número de preferências satisfeitas.

satisfeitas, os *stakeholders* com o mínimo e o máximo de preferências satisfeitas, a média delas e o desvio padrão.

Agora que já coletamos todas informações do arquivo, podemos analisar os resultados. Como o modelo utilizado não conta com uma grande quantidade de *features* e foram criadas apenas três configurações de *stakeholder* tivemos resultados parecidos para estratégias diferentes, e até mesmo recomendações iguais para algumas delas. Podemos observar que neste caso tivemos quatro estratégias (AVG, ANP, CR, MUL) que conseguiram satisfazer todas as oito restrições *soft* do grupo, alcançando o total de 2.2 no valor de satisfação de *stakeholder*. Nesses casos a configuração selecionada foi a que contém apenas as *features* F1, F2, F3, F7 e F8. Outras estratégias não tiveram resultados tão bons, e satisfizeram apenas sete das restrições (BC, LM, MP). O funcionamento de algumas estratégias fica evidente quando olhamos para alguns dados das tabelas, como por exemplo, a LM alcançou o maior valor possível no quesito satisfação mínima. Na MP o maior valor possível para o quesito satisfação máxima foi alcançado. A estratégia AVG obteve a maior média possível, e a ANP conseguiu a maior média na quantidade de preferências satisfeitas.

Não podemos afirmar qual estratégia se enquadra como ótima, pois dependendo do caso uma pode se tornar mais adequada para uma situação enquanto uma estratégia diferente pode ter melhor desempenho para outra situação. Os melhores resultados a serem alcançados variam de acordo com o objetivo do grupo a ser analisado, dessa forma uma melhor compreensão do funcionamento de uma determinada estratégia pode poupar o trabalho de executar o algoritmo para todas elas.

Neste capítulo detalhamos a implementação da ferramenta chamada SACRES 2.0, mostrando suas funcionalidades e arquitetura juntamente com o algoritmo para a recomendação de configurações, os elementos principais da sua interface e uma breve análise dos resultados obtidos para um exemplo simples.

5 TESTES DE DESEMPENHO

Neste capítulo apresentaremos alguns testes realizados para medir o desempenho do plug-in. Como mencionado anteriormente o problema atacado neste trabalho não é trivial e necessita de suporte computacional, então resolvemos executar alguns testes para medir a performance do algoritmo implementado. As tarefas testadas foram a leitura e construção do modelo de *features* pela ferramenta e o tempo de recomendação das configurações ótimas. Os testes foram executados em um computador com processador Intel Core i5-2320 3.00GHz e 6GB de memória RAM rodando Windows 7 64 bits.

Usamos três modelos de *features* para basear nossos testes. O primeiro foi apresentado anteriormente (na Figura 2.3) e foi utilizado como exemplo para as explicações da seção anterior. O segundo trata-se de um modelo representando algumas possibilidades na definição de um grafo, como podemos ver na Figura 5.1. O terceiro modelo, utilizado apenas para a primeira parte dos testes, é uma variação do modelo do grafo, porém maior e mais complexo, e está presente no conjunto de exemplos do FeatureIDE.

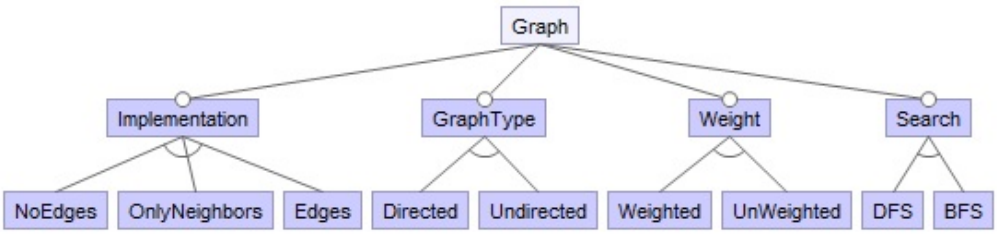


Figura 5.1: Modelo de *features* para um grafo.

A Tabela 5.1 mostra algumas informações referentes à quantidade de *features* de cada modelo. O primeiro modelo descrito no parágrafo anterior será chamado de Exemplo, o segundo de Grafo, e o terceiro de GPL. A última linha da tabela mostra a quantidade de *features* que não são obrigatórias presentes no modelo, ou seja, o conjunto de *features* as quais os *stakeholders* poderão expressar suas preferências sobre elas.

Quantidade de *features*	Exemplo	Grafo	GPL
Total	11	14	66
Concretas	10	13	36
Não-obrigatórias	9	13	53

Tabela 5.1: Quantidade dos tipos de *features* dos modelos.

5.1 Testes de preparação do modelo de *features*

Primeiramente foram realizados os testes referentes à preparação dos modelos para serem utilizados pela ferramenta. Como foi descrito no Capítulo 4, o plug-in executa duas tarefas antes de qualquer outra ação ser feita pela *view*: a leitura do arquivo no formato XML que representa o modelo de *features* e a construção do modelo dentro da ferramenta, para possibilitar a execução das demais funções incluindo a recomendação de configurações. Analisando os dados da Tabela 5.2 fica evidente que estas duas tarefas não causarão problemas no desempenho do plug-in, visto que o tempo necessário para a realização de ambas é muito pequeno. Foi constatado que o tempo consumido pela leitura e pela construção do modelo é bastante semelhante. Naturalmente, quanto maior o modelo, maior será o tempo gasto para as tarefas, porém esse aumento na duração não é muito significativo.

Tarefa	Exemplo	Grafo	GPL
Leitura	4ms	5ms	14ms
Construção	4ms	5ms	14ms

Tabela 5.2: Tempos de leitura e construção dos modelos pela ferramenta.

5.2 Testes de recomendação de configurações

A segunda fase de testes consistiu em medir o tempo gasto para a execução do algoritmo de recomendação de configurações ótimas. Como os modelos utilizados não são muito complexos, não medimos o tempo gasto por cada estratégia, mas sim do processo todo, desde o clique no botão com a função de executar o algoritmo até a obtenção dos resultados. Outro motivo para essa medição envolvendo todo o processo foi que grande parte do tempo é consumido no momento da geração do conjunto de consideração, antes da execução dos algoritmos das estratégias. Não foi incluso o tempo para a geração dos arquivos resultantes, por se tratar de uma parcela mínima do total gasto.

Na Tabela 5.3 estão os primeiros resultados a respeito do modelo Exemplo. O grupo de teste contava em média com quatro preferências de *stakeholder* por configuração. Cada etapa dos testes presentes neste capítulo foi executada três vezes, e a média do tempo gasto entre elas é o valor indicado nas tabelas seguintes. Incrementamos a quantidade de configuração a cada execução para analisar o aumento de tempo consumido. A primeira linha representa o caso exibido na Seção 4.4, onde tivemos três configurações com uma média de quatro preferências por configuração. Podemos observar que o aumento na quantidade de configurações a serem analisadas não causa um grande consumo adicional de tempo necessário para a recomendação.

Quantidade de configurações	Tempo gasto (ms)
3	44
6	54
10	60

Tabela 5.3: Testes com média de quatro preferências por configuração.

Resolvemos aumentar o número de preferências por configuração e medir novamente o tempo gasto. Na Tabela 5.4 estão os resultados das execuções com preferências definidas sobre todas *features* do modelo, que no caso do Exemplo somam um total de nove *features* não-obrigatórias (conforme a última linha da Tabela 5.1), ou seja, aproximadamente o dobro da quantidade utilizada no teste anterior. Como esperado para esses casos o tempo gasto foi maior devido ao aumento na quantidade de conflitos entre as preferências dos *stakeholders*, e podemos ver que o aumento no número de configurações continuou não ocasionando uma grande quantidade de tempo consumido.

Quantidade de configurações	Tempo gasto (ms)
3	183
6	201
10	209

Tabela 5.4: Testes com preferências para todas as *features*.

Por último realizamos medições em um modelo de *features* diferente do utilizado nos testes anteriores. A Tabela 5.5 apresenta os resultados referentes ao modelo chamado de Grafo, exibido na Figura 5.1. Neste caso foram definidas em média quatro preferências para cada configuração do grupo. É notável o aumento no consumo de tempo quando o modelo não é tão simples como o descrito no Exemplo, pois o modelo Grafo possui uma maior quantidade de grupos de *features* alternativas que necessitam a presença de um membro de cada grupo na configuração final, caso o nodo pai seja selecionado. Como foi apontado nos testes anteriores o aumento na quantidade de configurações não se reflete muito significativamente no tempo gasto, então para esse modelo excluímos o teste com dez configurações feito anteriormente.

Quantidade de configurações	Tempo gasto (ms)
3	394
6	415

Tabela 5.5: Testes utilizando o modelo Grafo da Figura 5.1.

5.3 Considerações Finais

Após a realização de todos os testes descritos neste capítulo concluímos que na nossa implementação o fator que mais afeta o tempo total necessário para a recomendação das configurações ótimas é a quantidade de preferências definidas pelos *stakeholders*. O aumento no número de configurações pertencentes ao grupo não ocasiona um gasto significativo no tempo consumido. Além disso, a complexidade do modelo de *features* afeta consideravelmente o tempo que será gasto para a geração do conjunto de consideração antes mesmo da execução dos algoritmos para cada estratégia.

6 CONCLUSÃO

Configuração de modelos de *features* é uma atividade importante para o desenvolvimento de produtos em uma Linha de Produto de Software. O envolvimento de múltiplos *stakeholders* nesse processo pode torná-lo difícil devido aos possíveis conflitos entre suas preferências. Neste trabalho apresentamos a ferramenta SACRES 2.0, um plug-in para o Eclipse que funciona de maneira integrada com o FeatureIDE, e que visa dar suporte à recomendação de configurações considerando múltiplos *stakeholders*. A ferramenta utiliza sete estratégias de escolha social para buscar configurações que melhor satisfaçam o grupo de *stakeholders*.

O foco deste trabalho não foi a análise de cada estratégia usada, e sim as funcionalidades fornecidas pela ferramenta. Detalhes mais aprofundados sobre o desempenho das estratégias, bem como uma avaliação do resultado das recomendações, podem ser encontrado em (STEIN; NUNES; CIRILO, 2014).

De maneira geral os resultados obtidos pela ferramenta foram satisfatórios. A integração com a interface do FeatureIDE através da nova *view* do SACRES 2.0 fornece uma interação intuitiva ao usuário e de fácil usabilidade. As funcionalidades de exibição de modelos de *features* e de configurações de *stakeholders* providas pelo FeatureIDE encaixaram perfeitamente nas etapas necessárias envolvendo todo o processo de recomendação.

Os testes realizados mostraram que não é necessário muito tempo para a obtenção de resultados ótimos para o problema. O aumento do tempo necessário é causado principalmente pela complexidade do modelo de *features* e pela quantidade de preferências que os *stakeholders* do grupo expressam em suas configurações.

Futuramente, possíveis melhorias podem ser adicionadas à ferramenta. Uma possibilidade seria considerar pesos para determinadas preferências dos *stakeholders* de acordo com sua expertise ou preferências sobre grupos de *features*.

Poderiam também ser realizados estudos de avaliação de usabilidade da ferramenta. Um possível teste contaria com grupos de *stakeholders* definindo preferências sobre um modelo de *features* e após a recomendação das configurações os usuários avaliariam os resultados obtidos. Estudos envolvendo cenários reais poderiam ser realizados de modo a validar a eficiência e usabilidade tanto do plug-in quanto da abordagem proposta, através de experimentos com os funcionários de uma empresa. Por se tratar de um plug-in para o Eclipse não seria muito complicado realizar possíveis extensões para este trabalho, seja adicionando funcionalidades diretamente no código do projeto ou criando um novo plug-in para realizar interações com os resultados gerados.

REFERÊNCIAS

APEL, S.; KÄSTNER, C. An Overview of Feature-Oriented Software Development. **Journal of Object Technology**, [S.l.], v.8, n.5, p.49–84, 2009.

CLEMENTS, P.; NORTHROP, L. **Software product lines**: practices and patterns. [S.l.]: Addison-Wesley Reading, 2002. v.59.

CZARNECKI, K.; EISENECKER, U. **Generative Programming**: methods, tools, and applications. [S.l.]: Addison Wesley, 2000.

KANG, K. C. et al. **Feature-oriented domain analysis (FODA) feasibility study**. [S.l.]: DTIC Document, 1990.

KRUEGER, C. Easing the transition to software mass customization. In: **Software Product-Family Engineering**. [S.l.]: Springer, 2002. p.282–293.

MACHADO, L. et al. SPLConfig: product configuration in software product line. In: BRAZILIAN CONGRESS ON SOFTWARE (CBSOFT), TOOLS SESSION. **Anais...** [S.l.: s.n.], 2014. p.1–8.

MENDONCA, M.; COWAN, D. Decision-making coordination and efficient reasoning techniques for feature-based configuration. **Science of Computer Programming**, [S.l.], v.75, n.5, p.311–332, 2010.

STEIN, J.; NUNES, I.; CIRILO, E. Preference-based feature model configuration with multiple stakeholders. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE-VOLUME 1, 18. **Proceedings...** [S.l.: s.n.], 2014. p.132–141.

THüM, T. et al. FeatureIDE: an extensible framework for feature-oriented software development. **Sci. Comput. Program.**, Amsterdam, The Netherlands, The Netherlands, v.79, p.70–85, Jan. 2014.

VAN DER LINDEN, F.; POHL, K. **Software Product Line Engineering**: foundations, principles, and techniques. [S.l.]: Stuttgart: Springer, 2005.