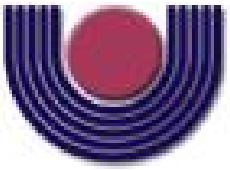


Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Avaliando uma Ferramenta para Migrar Código Legado para uma Linha de Produtos de *Software*: Um Estudo Experimental Preliminar

Gabriel Bruscatto

CASCAVEL
2016

Gabriel Bruscatto

Avaliando uma Ferramenta para Migrar Código Legado para uma Linha de Produtos de *Software*: Um Estudo Experimental Preliminar

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Ivonei Freitas da Silva

CASCAVEL
2016

Gabriel Bruscatto

AVALIANDO UMA FERRAMENTA PARA MIGRAR CÓDIGO LEGADO PARA UMA LINHA DE PRODUTOS DE *SOFTWARE*: UM ESTUDO EXPERIMENTAL PRELIMINAR

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Ivonei Freitas da Silva (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Victor Francisco Araya Santander
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Sidgley Camargo de Andrade
Colegiado de Sistemas para Internet, UTFPR -
Toledo

Cascavel, 6 de março de 2017

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

LPS	Linha de produtos de *software*
FOP	Programação orientada a *features*
AOP	Programação orientada a aspectos
DOP	Programação orientada a *delta*
COP	Programação orientada a contexto
VSoC	Separação virtual de interesses

Sumário

Resumo

Contexto: Uma linha de produtos de *software* (LPS) permite que produtos de *software* sejam construídos a partir de características comuns, diminuindo assim o tempo e esforço necessários para finalização de cada produto. Entretanto, a migração para uma abordagem de LPS por empresas de *software* que possuem sistemas legados pode ser dificultada por uma série de fatores, inclusive a falta de apoio ferramental.

Objetivo: Avaliar a ferramenta FeatureIDE em relação ao apoio por ela fornecido para desenvolvedores de *software* legado nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, quando utilizando uma abordagem de LPS.

Metodologia: Para avaliação da ferramenta FeatureIDE, um experimento foi conduzido envolvendo 18 estudantes de ciência da computação. Dois questionários, de *background* e *feedback*, foram utilizados para coleta da base de conhecimento dos estudantes e das suas opiniões a respeito do uso da ferramenta.

Resultados: O resultado do experimento indica que a FeatureIDE apoia desenvolvedores de *software* legado nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos.

Conclusões: As informações obtidas no experimento sugerem que a ferramenta FeatureIDE pode ser utilizada para apoiar empresas de tecnologia no processo de migração de sistemas legados para uma linha de produtos de *software*.

Palavras-chave: Linhas de Produtos de *Software*, Apoio Ferramental, Sistemas Legados

Capítulo 1

Introdução

Desde o início da programação, sistemas vêm sendo criados individualmente. Cada um envolvendo investimentos em análise de requisitos, arquitetura, *design*, documentação, etc. Porém, o mercado de tecnologia cada vez mais exige novos lançamentos em prazos de entrega menores. Assim, a capacidade de lançar produtos e serviços inovadores em um curto espaço de tempo tornou-se essencial para que empresas consigam manter-se ativas e crescentes. Mas, se sabe que produtos voltados a um mesmo domínio de aplicação compartilham características que podem ser reutilizadas a fim de se otimizar o processo de produção. Pensando nisso, foi desenvolvida a abordagem de linhas de produtos de *software* (LPS) [Clements e Northrop 2001].

De acordo com Cohen [Cohen 2002], a definição de LPS mais aceita na indústria é a de Clements e Northrop, a qual descreve uma linha de produtos de *software* como um conjunto de sistemas que utilizam *software* intensivamente, compartilhando um conjunto de características comuns e gerenciadas, que satisfazem as necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de um conjunto comum de ativos principais e de uma forma preestabelecida [Clements e Northrop 2001].

Como exemplo de sucesso, pode-se mencionar o estudo de caso feito por [Brownsword e Clements 1996] envolvendo o processo de implantação de uma linha de produção na CelsiusTech, empresa contratada pela marinha sueca para o desenvolvimento de sistemas para embarcações militares. Ao final da implantação, observou-se que a participação do *software* no custo total atingiu uma parcela de 20% em oposição a 65% para sistemas similares não adeptos de uma abordagem de linha de produtos. Além disso, notou-se maior precisão na previsão de custos e alta satisfação por parte dos clientes.

Apesar das vantagens fornecidas pelas linhas de produtos de *software*, novas empresas em geral (principalmente as pequenas e médias) não possuem recursos, tempo e conhecimento necessários para adoção de uma LPS. Graças a isso, hoje existem no mercado empresas com sistemas legados transformados em uma série de variantes as quais exigem processos de manutenção e evolução custosos e demorados. E, mesmo que essas empresas queiram migrar seus sistemas legados para LPS, grande esforço é necessário para, por exemplo, identificar e gerenciar variabilidade em seus sistemas. Assim, organizações que procuram a adoção de linhas de produtos de *software* encontram uma série de fatores que atrapalham o processo de migração, incluindo a carência de especialistas em LPS, custos de treinamento elevados, ausência de maturidade administrativa e a falta de apoio ferramental [Bastos et al. 2011]. Em alguns casos, empresas buscam ferramentas para aperfeiçoar o processo de desenvolvimento. Em outros, a quantidade de informações e as relações de dependência entre artefatos são complexas a ponto de necessitarem de uma ferramenta que possa apoiar a gestão dessa informação.

Nesse sentido, este trabalho tem como objetivo avaliar a ferramenta *open-source* FeatureIDE em relação ao apoio por ela fornecido para desenvolvedores de *software* legado nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, quando utilizando uma abordagem de LPS.

Capítulo 2

Sistemas Legados

2.1 Definição

Sistemas legados podem ser informalmente definidos como ”grandes sistemas de *software* os quais não se sabe como lidar, mas são vitais para uma organização” [Bennett 1995] ou ”qualquer sistema de informação que resiste significativamente à modificação e à evolução” [Brodie e Stonebraker 1995]. Alguns autores consideram ainda, como conceito para sistema legado: ”toda aplicação em produção”. A partir da entrada de um determinado sistema no processo de produção, o mesmo pode já estar ultrapassado, considerando as tecnologias adotadas e prazo para implementação.

Contudo, um sistema legado não necessariamente é construído sobre uma tecnologia antiga. A ausência de documentação adequada, por exemplo, pode dificultar substancialmente a manutenção e evolução de um *software* relativamente novo. Esse tipo de sistema geralmente constitue parte importante de uma organização e periodicamente necessita de novas funcionalidades que o adequem às mudanças de demanda e necessidades de clientes.

Capítulo 3

Linhas de Produtos de *Software*

3.1 Definição

De acordo com Cohen [Cohen 2002], a definição de LPS mais aceita na indústria é a de Clements e Northrop, a qual descreve uma linha de produtos de *software* como um conjunto de sistemas que utilizam *software* intensivamente, compartilhando um conjunto de características comuns e gerenciadas, que satisfazem as necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de um conjunto comum de ativos principais e de uma forma preestabelecida.

A abordagem de linha de produtos de *software* fornece uma forma de customização em massa através da construção de soluções individuais baseadas em uma série de componentes de *software* reutilizáveis. Ela permite o individualismo entre produtos de *software*, porém mantendo os benefícios da produção em massa. A necessidade pelo individualismo surge a partir dos diferentes requisitos sobre o *software* em relação às funcionalidades, plataformas de destino e propriedades não-funcionais, como desempenho e consumo de energia. Hoje em dia, fabricantes de carros, computadores e muitos outros produtos permitem que seus clientes configurem produtos conforme seus desejos antes mesmo da produção. Como dito, o processo de produção ainda possui as propriedades da produção em massa, onde os produtos são fabricados em grande escala, muitas vezes de forma automatizada, com base em peças padronizadas e reutilizáveis. No entanto, os fabricantes permitem variações dentro das etapas da produção, onde clientes podem optar por diferentes motores, cores e assim por diante, na compra de um carro, por exemplo. A Figura 3.1 apresenta a página de configuração de carros da BMW. Nela, o cliente pode escolher entre recursos disponíveis, incluindo potência, série e cor, de acordo

com suas necessidades.

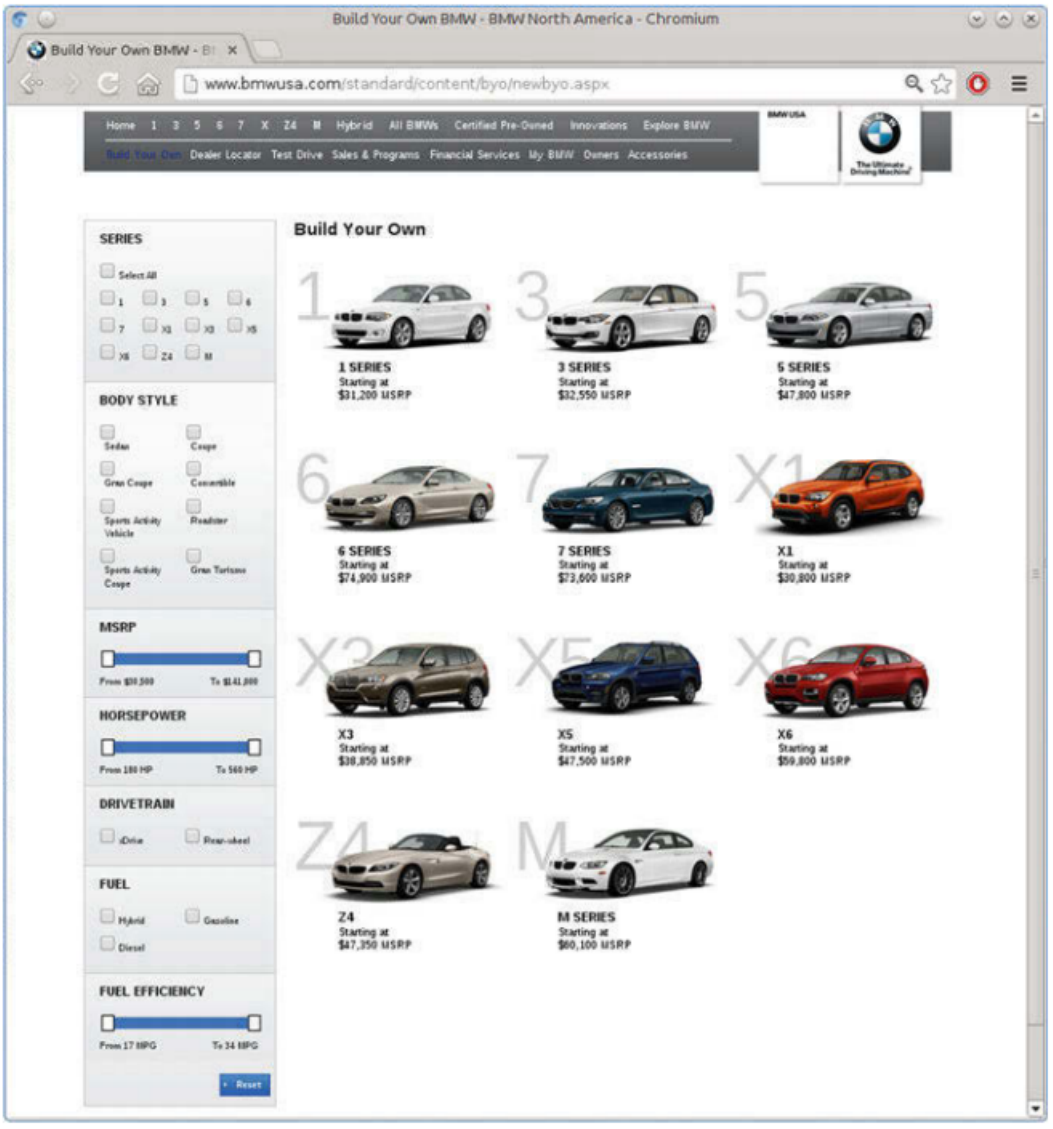


Figura 3.1: Página de configuração de carros da marca BMW [Apel et al. 2013]

Grande parte dos sistemas computacionais hoje contém um sistema operacional responsável pela comunicação entre *hardware* e *software*. Os sistemas operacionais atuais precisam atuar em diferentes plataformas e suprir as necessidades de várias aplicações. O *kernel* Linux, por exemplo, é executado em uma ampla variedade de plataformas, incluindo dispositivos embarcados, sistemas *desktop* e servidores, e suporta uma série de aplicações, desde programas de escritório e jogos, até aplicações de alta *performance*.

Porém, não há um único sistema operacional que apoie eficientemente todas as plataformas

e cenários de aplicação. Em vez disso, sistemas operacionais modernos são configuráveis. De fato, sistemas como o *kernel* Linux são linhas de produtos de *software*, onde usuários podem escolher entre um grande conjunto de opções para adaptar o *kernel* às suas necessidades. A Figura 3.2 apresenta uma das ferramentas de configuração do Linux, Kconfig, que pode ser usada para essa tarefa.

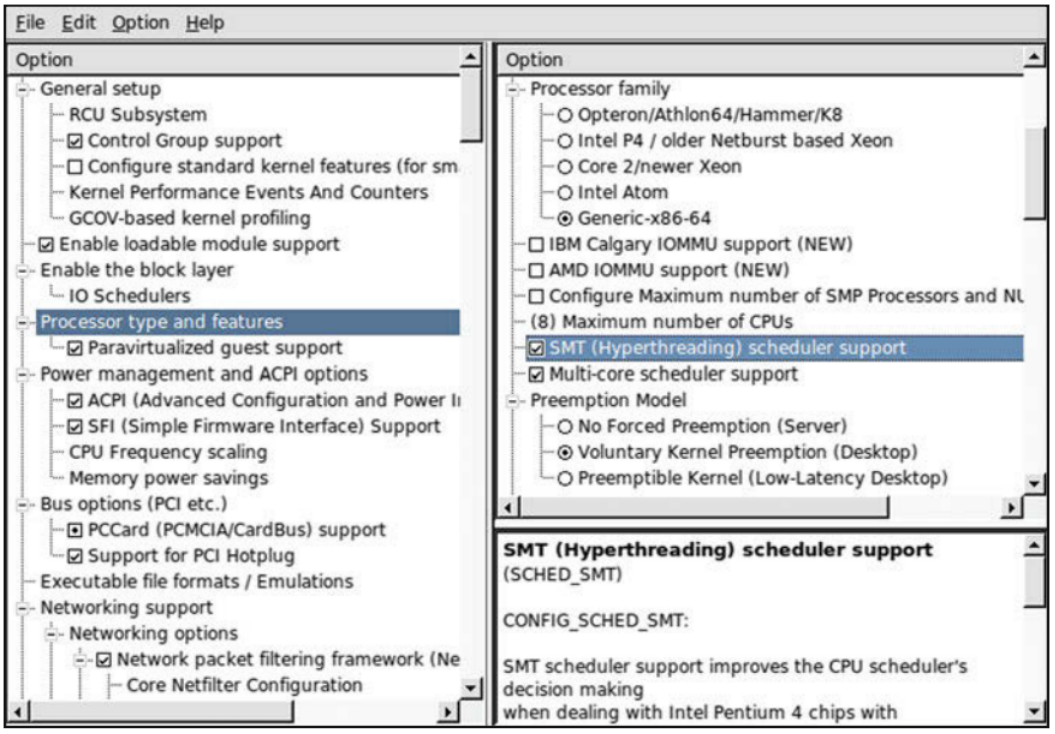


Figura 3.2: Ferramenta de configuração do *kernel* Linux [Apel et al. 2013]

3.2 Promessas das Linhas de Produtos de *Software*

As linhas de produtos de *software* apresentam uma série de vantagens em relação ao desenvolvimento individual de produtos de *software*, das quais as mais importantes estão apresentadas a seguir [Apel et al. 2013]:

- **Sob medida:** A abordagem de linhas de produtos no desenvolvimento de sistemas facilita a adaptação exclusiva de produtos para diferentes clientes. Em vez de fornecer um produto sem alternativas de modificação, ou com apenas algumas opções (edições *Community*, *Professional* e *Enterprise* por exemplo), um fornecedor de produtos de *software* pode produzir uma série de produtos sob medida.

- **Custos reduzidos:** Para vender a cada cliente sua solução desejada, não há a necessidade de se gastar com *design* e desenvolvimento a partir do zero. Em vez disso, são desenvolvidas partes reutilizáveis que podem ser combinadas de diferentes maneiras a fim de se gerar produtos distintos. Assim, o custo para venda de um novo produto se resume ao processo de escolha de quais partes previamente construídas serão adicionadas à solução e a realizar testes sobre o produto resultante. Porém, o investimento para uma abordagem como essa é certamente maior em relação ao desenvolvimento individual de software, visto que há a necessidade de se pensar no *design* e implementação dos diferentes módulos de forma que possam ser reutilizados mais tarde, mesmo que isso não seja necessário no primeiro produto. Mesmo assim, a longo prazo, a abordagem de linhas de produtos de *software* se torna vantajosa, conforme a demanda por produtos sob medida aumenta, como ilustra a Figura 3.3.

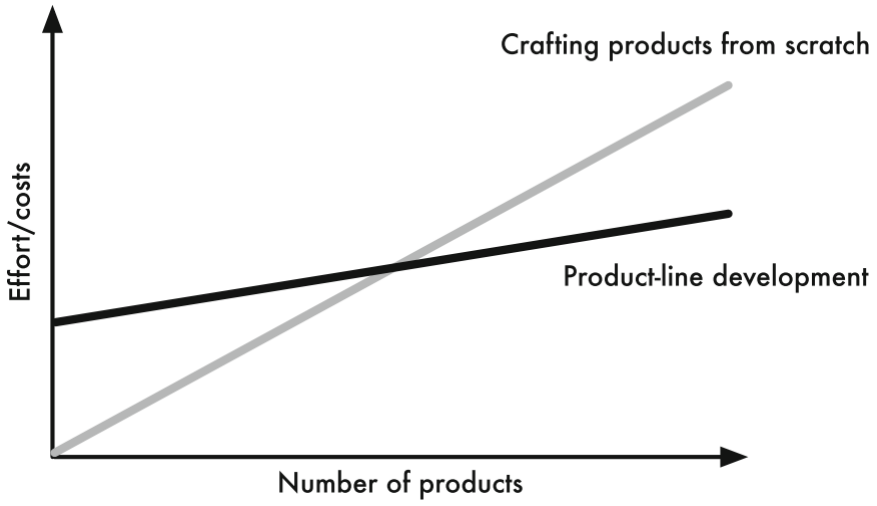


Figura 3.3: Esforço/custos do desenvolvimento de *software* a partir do zero *versus* abordagem de linhas de produtos de *software* [Apel et al. 2013]

- **Qualidade:** A produção industrializada em massa contribuiu para a qualidade dos produtos, visto que partes padronizadas podem ser checadas e testadas em diferentes produtos. Mesmo que nem todas as combinações de partes reusáveis sejam construídas, partes com uso frequente garantem produtos de *software* mais estáveis e confiáveis.

- **Prazo de comercialização:** Enquanto o *software* padronizado está prontamente disponí-

vel, produtos desenvolvidos individualmente requerem custos e tempo significativos antes que possam ser entregues. Quando um produto é gerado através de opções de configuração, ele pode ser gerado através da escolha e junção das partes pré-fabricadas.

3.3 Adoção de Uma Linha de Produtos de *Software*

A adoção de uma linha de produtos de *software* pode acontecer de diversas formas. Em grande parte dos casos, uma empresa já possui alguns produtos do domínio. Com isso, deve-se pensar não em um desenvolvimento a partir do zero, e sim em um processo de migração para uma abordagem de linha de produtos. Existem três caminhos distintos para o processo de adoção [Krueger 2002]:

- **Abordagem proativa:** Busca desenvolver uma linha de produtos a partir do zero, através da utilização cuidadosa de métodos de análise e *design*.
- **Abordagem extrativa:** Conta, inicialmente, com uma coleção de produtos existentes e os refatora para uma linha de produtos.
- **Abordagem reativa:** Inicia com uma linha de produtos pequena e fácil de manipular, podendo ela ser formada inclusive por um único produto, e a estende com novas *features* e artefatos de implementação, expandindo assim o escopo da linha.

Graças à necessidade de alto investimento inicial, tempo e nível de conhecimento, grande parte das organizações não optam pela abordagem proativa. Assim, uma abordagem bastante utilizada para adoção de uma LPS consiste em um processo de migração de sistemas tipicamente construídos para uma linha de produtos (abordagem extrativa).

3.4 Conceitos Relacionados

As linhas de produtos de *software* são teoricamente amplas e envolvem uma série de conceitos importantes e necessários para o entendimento do tema. Os conceitos utilizados neste trabalho são essenciais para o entendimento do mesmo e estão descritos a seguir.

3.4.1 *Feature*

Segundo [Apel et al. 2013], uma *feature* é uma característica ou um comportamento visível pelo usuário final de um sistema de *software*. *Features* são usadas na engenharia de linhas de produtos para especificar e comunicar semelhanças e diferenças dos produtos entre *stakeholders*, e para orientar a estrutura, o reuso e as variações de componentes ao longo de todas as fases do ciclo de vida de um *software*. Ainda no contexto de linhas de produtos, *features* são utilizadas para distinguir produtos de uma linha de produtos, por exemplo:

- "O aplicativo a ser desenvolvido deve funcionar tanto em Android quanto em iOS."
- "Ambas as aplicações financeiras suportam transações internacionais."
- "O reprodutor de vídeos A suporta o formato .MP4, o reprodutor de vídeos B não."

3.4.2 Abordagem Orientada a *Features*

Linhas de produtos de *software* facilitam a industrialização do desenvolvimento. Baseando-se em um conjunto de peças reutilizáveis, um fabricante de *software* pode gerar um produto com base nos requisitos de um cliente. Assim, o conceito de *feature* é fundamental para atingir esse nível de automação.

A ideia principal da orientação a *features* é organizar e estruturar o processo de linha de produção, assim como todos os artefatos envolvidos, em termos de funcionalidades. Dessa maneira, torna-se mais fácil rastrear os requisitos de um determinado cliente para os artefatos de *software* capazes de prover a funcionalidade desejada. Em termos técnicos, pode-se dizer que a abordagem orientada a *features* explicita as *features* nos requisitos, código, testes, e assim por diante, por todo o ciclo da linha de produção.

3.4.3 Interação Entre *Features*

A ideia principal da orientação a *features* é fazer com que elas estejam explícitas no *design* e código, tanto através da anotação de trechos de código relacionados a cada funcionalidade, quanto pela separação e modularização dos mesmos. Porém, *features* não atuam de maneira isolada e independente. Elas interagem de diversas formas, seja de maneira positiva e planejada, ou de forma crítica e inadvertida. Em geral, espera-se que *features* interajam e com isso

sejam capazes de trocar informações, refinar o comportamento de outras *features*, reusar funcionalidades e concluir tarefas cooperativamente. Todavia, interações não planejadas podem gerar anomalias e resultar em estados críticos do sistema. Assim, especificar e gerenciar as interações planejadas, e detectar e resolver as não desejadas é um dos maiores desafios do desenvolvimento de linhas de produtos de *software* orientado a *features*. A Figura 3.4 ilustra um exemplo de interação entre três *features*. Nela, é possível visualizar quatro interações: Três delas envolvendo duas *features*, e uma delas envolvendo três.

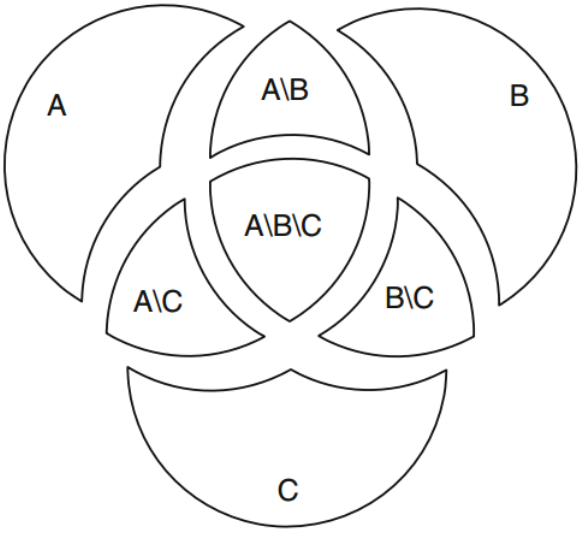


Figura 3.4: Visualização da interação entre as três *features* A, B e C [Apel et al. 2013]

3.4.4 Diagrama de *Features*

Um diagrama de *features* é uma notação visual utilizada para especificação de um modelo de *features*. Consiste de uma árvore onde os nós representam os nomes de cada *feature*. Se uma *feature* f é filha de outra *feature* p, por exemplo, então f pode ser selecionada somente quando p também é. Normalmente, esse tipo de diagrama inclui relações mútuas. Por exemplo, a *feature* pai denota um conceito mais geral, enquanto a filha representa uma especialização.

Features obrigatórias e opcionais são diferenciadas por um círculo presente no nó filho: quando vazio, o círculo representa uma *feature* opcional e, quando preenchido, indica obrigatoriedade, assim como ilustrado na Figura 3.5. A relação de obrigatoriedade exige que, quando a *feature* pai é selecionada, a *feature* filha também seja incluída, o que não é obrigatório na

relação de opcionalidade.

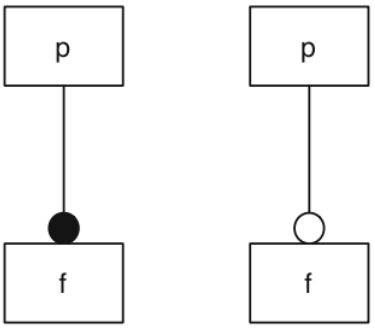


Figura 3.5: Notação para *features* obrigatórias (círculo preenchido) e opcionais (círculo vazio) [Apel et al. 2013]

A Figura 3.6 apresenta uma *feature* pai e um grupo de *features* filhas ligados por um arco vazio. Esse elemento gráfico indica a escolha de exatamente uma das *features* do conjunto.

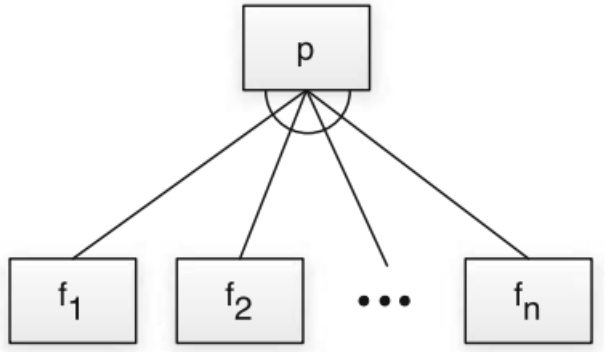


Figura 3.6: Notação para a escolha exclusiva, correspondente ao operador xor [Apel et al. 2013]

A Figura 3.7 mostra *features* filhas conectadas a uma *feature* pai através de um arco preenchido. Com isso, assume-se que há a possibilidade de escolha de uma ou mais *features* filhas. Ou seja, pode-se escolher pelo menos uma delas, mas não há um limite máximo.

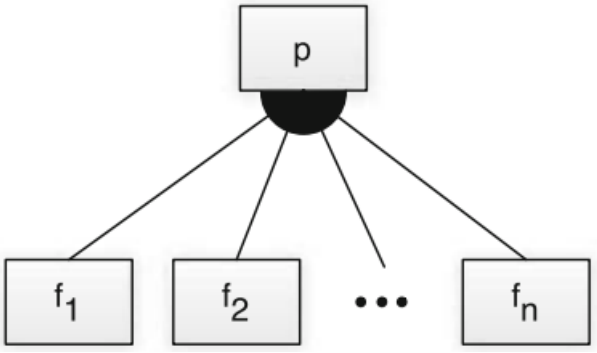


Figura 3.7: Notação para a escolha de uma ou mais *features* filhas, correspondente ao operador or [Apel et al. 2013]

3.4.5 Programação Orientada a *Features* (FOP)

É uma abordagem baseada em composição para construção de linhas de produtos de *software* que depende diretamente do conceito de *feature*. Sua ideia é decompor o *design* de um sistema e seu código em *features* de forma que a estrutura do sistema se alinhe com as mesmas. Para isso, são necessários novos recursos de linguagem, tanto para relacionar trechos de um programa a suas *features* correspondentes, quanto para permitir que o código seja encapsulado em unidades modulares e possíveis de serem combinadas. A Figura 3.8 ilustra o mapeamento entre *features* e artefatos de implementação na programação orientada a *features*.

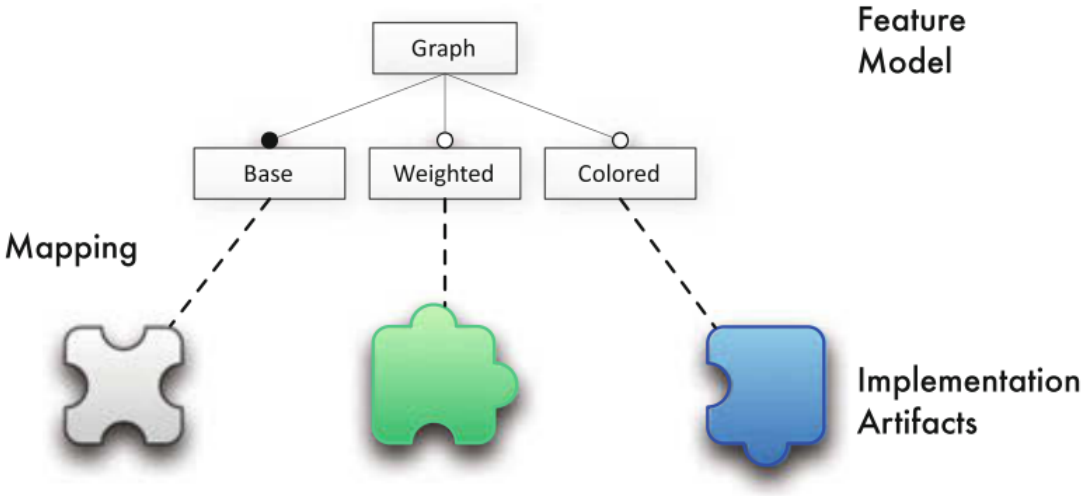


Figura 3.8: Mapeamento entre *features* e artefatos de implementação da programação orientada a *features* [Apel et al. 2013]

3.4.6 Programação Orientada a Aspectos (AOP)

Pode-se perceber na Figura 3.9 como a implementação da *feature* Colored (em destaque) está espalhada ao longo das três classes e, dentro delas, afeta dois métodos. Além disso, é possível notar que há replicação de trechos de código nas classes Node e Edge. A programação orientada a aspectos busca reduzir essas dispersões, emaranhamentos e replicações de código geradas pela má separação de interesses, ou seja, pela má separação dos trechos de código-fonte de acordo com suas devidas funcionalidades.

```
1  class Graph { /* ... */ }
2  class Node {
3    Color color = new Color();
4    Color getColor() { return color; }
5    int id = 0;
6    Node(int _id) { id = _id; }
7    void print() {
8      Color.setDisplayColor(getColor());
9      System.out.print(id);
10   }
11 }
12 class Edge {
13   Color color = new Color();
14   Color getColor() { return color; }
15   Node a, b;
16   Edge(Node _a, Node _b) { a = _a; b = _b; }
17   void print() {
18     Color.setDisplayColor(getColor());
19     System.out.print(" (");
20     a.print();
21     System.out.print(", ");
22     b.print();
23     System.out.print(") ");
24   }
25 }
26 class Color {
27   static void setDisplayColor(Color c) { /* ... */ }
28 }
```

Figura 3.9: Código da *feature* Colored disperso e replicado (em destaque) [Apel et al. 2013]

3.4.7 Programação Orientada a *Delta* (DOP)

A ideia da programação orientada a *delta* está relacionada ao AOP e FOP. Um programa consiste de um módulo base e um conjunto de módulos *delta* que modificam o módulo base gradativamente. Essa assimetria entre os dois tipos de módulos é semelhante ao AOP, onde se distingue o programa base dos aspectos, e ao FOP, o qual diferencia classes e seus refinamentos. Assim como em um módulo de *features*, um módulo *delta* pode adicionar novas classes e membros bem como estender os métodos existentes através de substituição. Em contraste com os módulos de *features*, os módulos *delta* têm a capacidade de excluir classes existentes e membros individuais. A Figura 3.10 ilustra a utilização de um módulo *delta* para substituição

do método print por outro método com nome display.

O fato de que módulos *delta* podem excluir elementos do programa gera uma série de implicações. Uma delas é que módulos *delta* podem encolher o programa base, o que não acontece com módulos de *features*. Porém, um dos problemas do DOP é que a expressividade adicional presente pode causar problemas referentes à manutenção e compreensão do programa. Olhando para um programa, pode ser difícil concluir se alguns módulos *delta* irão eliminar um elemento necessário, por exemplo.

```
1 core BasicGraph {
2   class Graph { /*...*/ }
3   class Node {
4     int id = 0;
5     Node(int _id) { id = _id; }
6     void print() { System.out.print(id); }
7   }
8   class Edge { /*...*/ }
9 }
```

```
1 delta IdToName when BasicGraph {
2   modifies class Node {
3     removes void print();
4     adds void display() { System.out.print(id); }
5   }
6 }
```

Figura 3.10: Usando um módulo *delta* para substituição do método print pelo método display [Apel et al. 2013]

3.4.8 Programação Orientada a Contexto (COP)

A ideia da programação orientada a contexto é que a linguagem de programação deve apoiar o programador a expressar o comportamento dependente de contexto de um programa [Hirschfeld, Costanza e Haupt 2008]. Dependendo do contexto atual, o qual pode alterar dinamicamente em tempo de execução, o comportamento do programa pode ser diferente. Contexto é cada peça de informação que é externa ao programa, como o sistema operacional, o desempenho da rede, a localização física do dispositivo e assim por diante. Tecnicamente, linguagens orientadas a contexto usam uma série de mecanismos para implementar os diferentes comportamentos dependentes de contexto, assim como condições específicas ou estáticas.

A Figura 3.11 mostra a implementação da *feature* Colored usando a linguagem orientada a

contexto ContextJ [Appeltauer et al. 2011].

```
class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() { /* ... */ }
  layer Colored {
    Color color = new Color();
    Color getColor() { return color; }
    before void print() {
      Color.setDisplayColor(getColor());
    }
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() { /* ... */ }
  layer Colored {
    Color color = new Color();
    Color getColor() { return color; }
    before void print() {
      Color.setDisplayColor(getColor());
    }
  }
}
```

Figura 3.11: Implementação da *feature* Colored como uma camada dependente de contexto na linguagem ContextJ [Apel et al. 2013]

3.4.9 Variabilidade

Variabilidade [Apel et al. 2013] é a habilidade de se derivar vários produtos a partir de um conjunto comum de artefatos. O desejo de se construir *software* a partir de variabilidade é impulsionado pelo espectro normalmente grande de exigências vindas de *stakeholders* em uma linha de produção, as quais se manifestam através de *features* almejadas no produto final. Assim, ao invés de se satisfazer as particularidades de um cliente individualmente e a partir do zero, o *software* com variabilidade pode ser configurado aos novos requisitos, de forma que o esforço necessário para fabricação do produto seja reduzido substancialmente.

3.4.10 Variante e Ponto de Variação

É o termo utilizado para se referir a uma variante particular de uma *feature* variante. Uma única variante pode consistir de várias entidades de *software*, colaborando para garantir a fun-

cionalidade necessária na *feature* variante [Svahnberg, Gurp e Bosch 2005].

Ponto de variação diz respeito a determinados locais de um sistema de *software* onde são feitas escolhas sobre qual variante utilizar. Uma *feature* variante se transforma em um conjunto de variantes e pontos de variação em um sistema de *software*, e esses pontos de variação são utilizados para combinar uma variante particular com o resto do sistema [Svahnberg, Gurp e Bosch 2005].

3.5 Processo Para o Desenvolvimento de Linhas de Produtos de *Software*

Grande parte dos processos da engenharia tradicional de desenvolvimento de *software* mantém foco exclusivo em um único produto a ser construído. Etapas como coletas de requisitos, *design* e implementação do sistema não levam em consideração sistemas semelhantes que podem surgir futuramente. Em linhas de produtos de *software*, há uma preocupação relacionada ao desenvolvimento de uma série de produtos não idênticos, mas semelhantes em vários aspectos. Assim, um dos principais fatores de sucesso para o desenvolvimento de uma linha de produtos é a definição de um foco adequado em um domínio bem definido.

Uma das principais características das linhas de produtos de *software* é a separação entre a engenharia de domínio, engenharia de aplicação, espaço do problema e espaço de solução. A Figura 3.12 ilustra um processo de engenharia para o desenvolvimento de linhas de produtos de *software* proposto por [Apel et al. 2013], o qual será explicado em seguida.

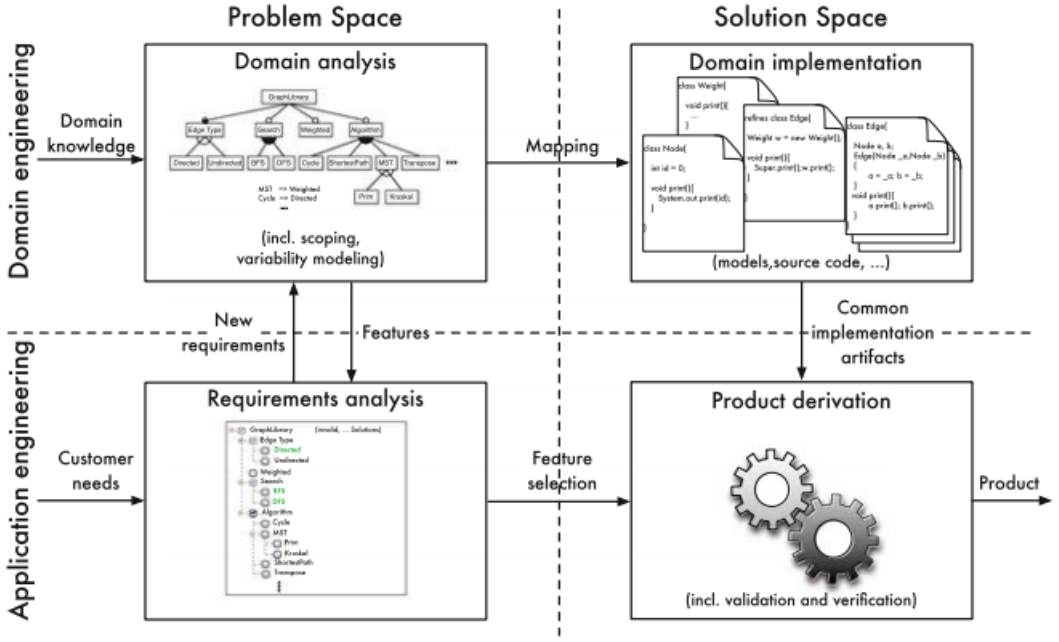


Figura 3.12: Processo de engenharia para linhas de produtos de *software* [Apel et al. 2013]

A engenharia de domínio é o processo de analisar o domínio de uma linha de produtos e desenvolver artefatos reutilizáveis. Ela não resulta em um produto de *software* específico, mas prepara os artefatos para que possam ser utilizados em múltiplos ou até mesmo todos os produtos. Assim, a engenharia de domínio foca no desenvolvimento voltado ao reuso. Por outro lado, a engenharia de aplicação possui o objetivo de desenvolver um produto específico para as necessidades de um cliente em particular. Corresponde ao processo de desenvolvimento tradicional de *software*, porém reaproveitando os artefatos da engenharia de domínio, quando possível. A engenharia de aplicação é repetida para cada produto da linha de produtos que será derivado.

A distinção entre o espaço de problema e o espaço de solução destaca duas diferentes perspectivas. O espaço de problema assume a perspectiva das partes interessadas e seus problemas, necessidades e pontos de vista em relação ao domínio como um todo e aos produtos individuais. Em contraste, o espaço de solução representa as perspectivas do fornecedor e desenvolvedor de *software*. Ele abrange a concepção, implementação, validação e verificação de *features* e suas combinações adequadas para facilitar o reuso.

As diferenças entre a engenharia de domínio e aplicação, bem como os espaços do problema

e de solução dão origem a quatro grupos de tarefas no desenvolvimento de linhas de produtos:

- **Análise de domínio:** É um modelo da engenharia de requisitos para uma linha de produtos. Aqui, é preciso definir o escopo do domínio, ou seja, decidir quais produtos devem ser englobados pela linha de produtos e, consequentemente, quais *features* são relevantes e devem ser implementadas como artefatos reutilizáveis.

- **Análise de requisitos:** Investiga as necessidades de um cliente específico como parte da engenharia de aplicação. Em casos mais simples, os requisitos do cliente são mapeados para uma escolha de *features* com base nas *features* identificadas durante a análise de domínio. Caso novos requisitos sejam descobertos, eles podem retornar à análise de domínio, o que possivelmente resultará em modificações no modelo de *features*.

- **Implementação do domínio:** É o processo de desenvolvimento de artefatos reutilizáveis que correspondem às *features* identificadas na análise de domínio. Dependendo de como a variabilidade é implementada, os desenvolvedores podem produzir diferentes artefatos nessa etapa, desde parâmetros de tempo de execução e diretivas de pré-processamento, até *plug-ins* e componentes em geral.

- **Derivação de produtos:** Também chamada de geração, configuração ou montagem de produtos, é a etapa de produção da engenharia de aplicação, onde artefatos reutilizáveis são combinados de acordo com os resultados da análise de requisitos. Dependendo da abordagem de implementação, esse processo pode ser mais ou menos automatizado, possivelmente envolvendo várias tarefas de desenvolvimento e customização.

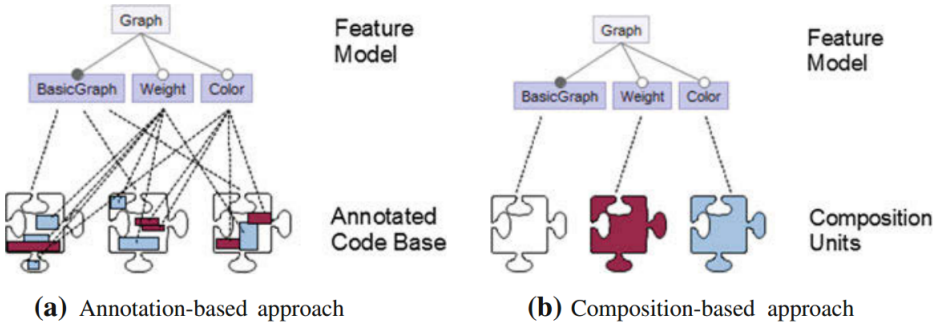
3.6 Abordagens para Implementação de Linhas de Produtos de *Software*

O principal objetivo da engenharia de linha de produtos orientada a *features* é a capacidade de se gerar produtos automaticamente a partir do código variável, de acordo com as *features* escolhidas pelo usuário. Consequentemente, a derivação de produtos envolve geração de produtos, estática ou dinamicamente. A seguir são descritas duas abordagens muito utilizadas na prática para implementação de linhas de produtos de *software*, as quais diferem na forma de representar variabilidade em código e de gerar produtos [Apel et al. 2013].

- **Abordagem baseada em anotação:** Realiza anotações em um código base, de tal forma que trechos de código pertencentes a uma *feature* estejam marcados adequadamente. Assim, durante a derivação de um produto, todo o código pertencente a *features* não selecionadas ou a combinações inválidas de *features* é removido em tempo de compilação ou ignorado em tempo de execução, para que o produto final seja formado. Essa abordagem é muito utilizada graças à facilidade de uso e ao suporte nativo presente em diversos ambientes de programação.

- **Abordagem baseada em composição:** Implementa *features* como unidades compostas, idealmente uma unidade por *feature*. Durante a derivação de produtos, todas as unidades pertencentes a *features* selecionadas e combinações válidas de *features* são compostas para que o produto final seja gerado. O principal desafio dessa abordagem é manter o mapeamento entre *features* e unidades de composição simples e maleável, em uma relação de um para um, idealmente. Se cada *feature* possui sua própria implementação sob a forma de uma unidade de composição, um gerador pode simplesmente incluir a unidade correspondente na composição quando uma *feature* é selecionada.

Uma forma de se visualizar a diferença entre as duas abordagens é o fato de que a abordagem baseada em anotação suporta variabilidade negativa, ou seja, trechos de código podem ser removidos quando necessário, enquanto a abordagem baseada em composição suporta variabilidade positiva, permitindo que unidades de composição sejam adicionadas sob demanda. A Figura 3.13 ilustra as diferenças entre as duas abordagens na implementação de uma LPS.



(a) Annotation-based approach **(b)** Composition-based approach

Figura 3.13: Abordagens baseadas em anotação e composição para implementação de uma linha de produtos [Apel et al. 2013]

3.7 Considerações Finais

As definições e conceitos apresentados neste capítulo são importantes para o entendimento da nomenclatura utilizada e do contexto geral do experimento. A abordagem para implementação de linhas de produtos de *software* utilizada, em conjunto com a ferramenta avaliada durante o experimento, é a abordagem baseada em anotação. Mais detalhes serão apresentados nos capítulos seguintes.

Capítulo 4

Escolha de Instrumentos para o Experimento

4.1 Escolha da Ferramenta

4.1.1 Ferramentas Candidatas

As ferramentas candidatas a participar do experimento foram as citadas por [Apel et al. 2013]. A Tabela 3.1 apresenta tais ferramentas e suas respectivas categorias, técnicas para gestão de variabilidade utilizadas e linguagens onde são aplicadas. As ferramentas marcadas com o símbolo * não são *open-source*.

4.1.2 Critérios para Escolha da Ferramenta

Considerando que o objetivo do estudo é a avaliação de uma ferramenta que possa ser utilizada por empresas de *software*, as quais diferem tanto em linguagem quanto em processos de desenvolvimento, para escolha da ferramenta os seguintes critérios foram usados:

- ***Open-source*:** Por se tratar de um experimento que busca avaliar a viabilidade da utilização de uma ferramenta *open-source* que possa apoiar empresas de tecnologia na migração para uma linha de produtos de *software*, é necessário que a ferramenta escolhida seja de código aberto.

- **Completude de técnicas e funcionalidades:** Um processo de linha de produtos envolve uma série de etapas como análise de domínio, análise de requisitos, implementação do domínio e derivação de produtos, como visto na Seção 3.5. Assim, a propriedade considerada mais importante neste trabalho para escolha de uma ferramenta que venha apoiar

o processo de migração para uma LPS é capacidade de trabalhar com diferentes abordagens de implementação de linhas de produtos de *software* (completude de técnicas) e de auxiliar maior parte ou todas as tarefas presentes nas etapas de uma LPS (completude de funcionalidades).

Tabela 4.1: Ferramentas candidatas

Ferramenta	Categoria	Técnica	Linguagem
pure::variants*	Modelagem de *features*, checagem de consistência, configuração, geração de produtos, ambiente de desenvolvimento	Pré-processamento	-
Gears*	Modelagem de *features*, checagem de consistência, configuração, geração de produtos, ambiente de desenvolvimento	Pré-processamento	-
FeatureIDE	Modelagem de *features*, configuração, checagem de consistência, prova de teoremas, métricas de código, ambiente de desenvolvimento, múltiplas linhas de produtos	Pré-processamento, FOP, AOP, DOP	Java, C, C++, C#, JML, Haskell, XML, Python, Alloy, Featherweight Java, JML, JCop, Stratego, SDF, JavaCC
AHEAD	Modelagem de *features*, configuração, geração de produtos	FOP	Java
Antenna	Geração de produtos	Pré-processamento	Java
AspectJ	Geração de produtos	AOP	Java
CDL	Modelagem de variabilidade	-	-
CIDE	Modelagem de *features*, configuração, geração de produtos, decomposição orientada a *features*, ambiente de desenvolvimento, checagem de tipos, métricas de código	VSoC	Featherweight Java, Java, C, C#, JavaScript, Haskell, Bali, ANTLR, JavaCC, Properties, HTML, XML, XHTML, XML-People, Python, OSGi Manifest
Clafer	Modelagem de *features,* modelagem de classes, modelagem de objetos, configuração	-	-
ConcernMapper	Mapeamento de *features*	-	Java
ContextJ	Geração de produtos	COP	Java
Cpp	Geração de produtos	Pré-processamento	C, C++
CVL	Modelagem de variabilidade	-	-
DeltaJ	Modelagem de *features*, configuração, geração de produtos, checagem de tipos	DOP	Java
FAMA	Modelagem de *features*, checagem de consistência, configuração	-	Java
FeatureHouse	Geração de produtos	FOP	Featherweight Java, C, C#, JML, Haskell, XML, Python, Alloy, Featherweight Java, JML, JCop, Stratego, SDF, JavaCC
FeatureMapper	Modelagem de *features*, checagem de consistência, configuração	-	Java
Git	Versionamento	-	-
Kbuild	Geração de produtos	-	-
Kconfig	Configuração	-	-
Koala	Geração de produtos	-	-
Munge	Geração de produtos	Pré-processamento	Java
OSGi framework	Soluções baseadas em componentes	-	Java
SNIP	Checagem de modelos	Pré-processamento	Promela
SPLOT	Modelagem de *features*, checagem de consistência, configuração	-	Java
SPLverifier	Teste, checagem de modelos	FOP	Java, C
TypeChef	Checagem de tipos	Pré-processamento	C
Undertaker	Checagem de consistência, amostragem	Pré-processamento	-
VMC	Checagem de modelos	Pré-processamento	Modal Transition System

- **Abrangência em linguagens:** Muitas linguagens são hoje utilizadas tanto no desenvolvimento quanto na manutenção e evolução de *software* comercial. Pontanto, espera-se que a ferramenta escolhida possa ser aplicada a uma grande quantidade de linguagens, de forma a se maximizar a gama de empresas capazes de usufruir das funcionalidades da ferramenta.

4.1.3 Ferramenta Escolhida

As ferramentas pure::variants e Gears foram excluídas da seleção inicialmente pelo fato de não serem *open-source*. Embora a ferramenta CIDE [Kästner, Apel e Kuhlemann 2008] alcance duas linguagens a mais, a Tabela 3.1 mostra que a ferramenta FeatureIDE [Apel, Leich e Marnitz 2005], suporta um número maior de técnicas. Portanto, FeatureIDE foi a ferramenta escolhida para avaliação.

4.1.4 FeatureIDE: Uma Ferramenta *Open-Source* para a Implementação de Linhas de Produtos

Enquanto as ferramentas comerciais fornecem soluções de qualidade para o desenvolvimento de linhas de produtos, pesquisadores e educadores podem procurar soluções de código aberto, de forma a se facilitar experimentações e utilizações em sala de aula. FeatureIDE é um ambiente de desenvolvimento *open-source* para linhas de produtos voltado principalmente para pesquisadores, professores e estudantes em geral. A ferramenta pode ser utilizada através do mecanismo de *plug-ins* do ambiente de desenvolvimento Eclipse.

Para a análise de domínio, a FeatureIDE disponibiliza um editor gráfico para a modelagem de *features*, como mostra a Figura 4.2. A geração de gráficos de alta qualidade dos modelos de *features* tem prioridade sobre a escalabilidade do editor gráfico, auxiliando assim em atividades como ensino e publicações de pesquisa, por exemplo. Além disso, o suporte para anotações adicionais e parâmetros booleanos é restrito comparado às ferramentas comerciais. A Figura 4.1 apresenta a sintaxe de anotação de *features* na ferramenta FeatureIDE.

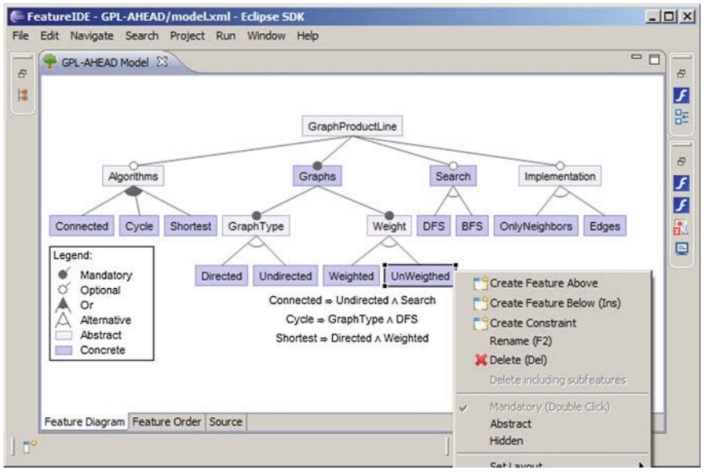


Figura 4.1: Editor de modelos de *features* da ferramenta FeatureIDE [Apel et al. 2013]

```
    //#if Reason2
//@     String Reason2 = "Mensagem de erro 2";
    //#endif

    //#if PrintReason
        errorMessage.printReason();
    //#endif
```

Figura 4.2: Sintaxe de anotação de *features* na ferramenta FeatureIDE [Apel et al. 2013]

A Figura 4.3 apresenta uma das telas da ferramenta da FeatureIDE, utilizada para visão detalhada sobre a linha de produtos, onde o usuário tem a oportunidade de observar *features* e o relacionamento entre elas.

O processo de configuração de produtos na ferramenta se dá através de arquivos de configuração disponíveis na pasta ”configs” dentro do projeto, onde é possível marcar as *features* a serem adicionadas ao produto final. Os arquivos são atualizados automaticamente, conforme funcionalidades são incluídas ou removidas da árvore de *features*. A Figura 4.4 mostra a configuração de produtos e a visualização da árvore de *features* na ferramenta.

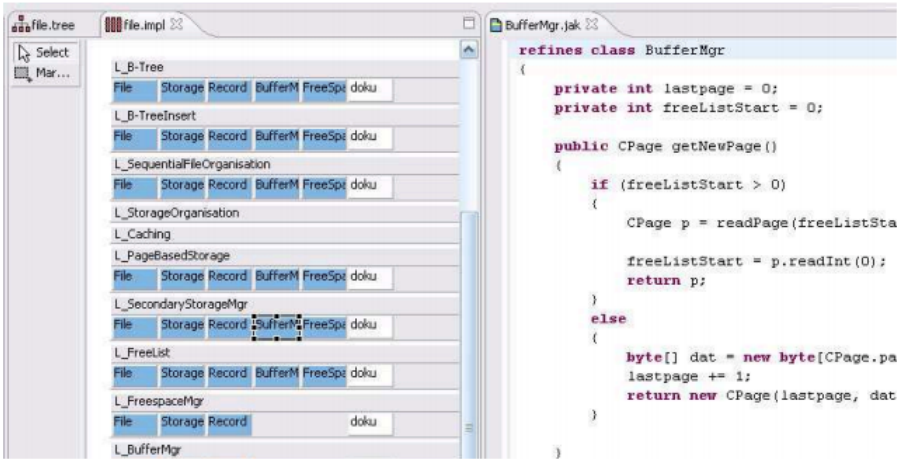


Figura 4.3: Visão detalhada da linha de produtos na ferramenta FeatureIDE [Apel, Leich e Marnitz 2005]

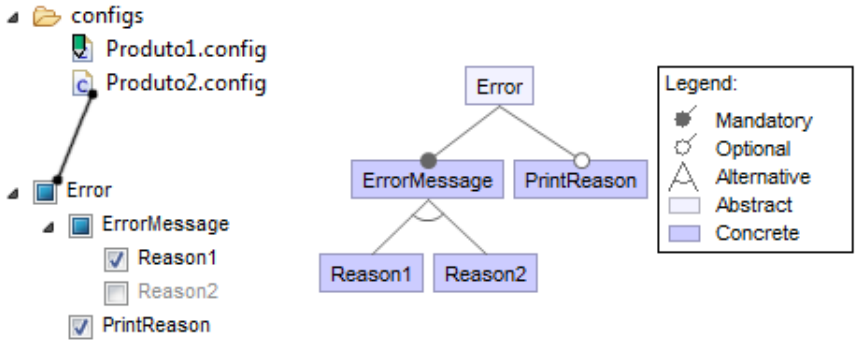


Figura 4.4: Configuração de produtos e visualização da árvore de *features* na ferramenta FeatureIDE

Para o espaço de solução, a FeatureIDE suporta várias ferramentas específicas como FeatureC++ [Apel et al. 2005] e FeatureHouse [Apel, Kästner e Christian 2009]. A FeatureIDE é extensível por *plug-ins* e fornece uma base importante para o ensino de linhas de produtos de *software*, como modelos de *features* e pré-processadores, por exemplo. Entretanto, em contraste com ferramentas comerciais, a FeatureIDE não provê mecanismos que lhe permita integração com outras ferramentas de forma arbitrária. Neste contexto, o seu desafio é a sua adoção em escala industrial.

4.2 Escolha do Código-fonte

A escolha do código-fonte para utilização no experimento foi baseada nas características da ferramenta escolhida e do objetivo do trabalho. Foi feita uma busca no site GitHub, até que um código-fonte adequado fosse encontrado. Mais detalhes a respeito dos critérios utilizados para escolha dos códigos-fonte e o código escolhido serão apresentados a seguir.

4.2.1 Critérios para Escolha dos Códigos-fonte

Partindo do princípio que o experimento envolveu a utilização da ferramenta FeatureIDE e o foco do trabalho envolve sistemas legados, os seguintes critérios foram definidos:

- **Linguagem compatível com a FeatureIDE:** Uma característica necessária para os trechos de código a serem utilizados no experimento é a linguagem. Esta deve estar entre as linguagens englobadas pela ferramenta.
- **Originado de *software* legado:** Para atender ao objetivo do trabalho, o código escolhido deve ser proveniente de *software* legado, ou seja, deve ser antigo e/ou estar inadequadamente documentado e/ou ser de difícil manutenção e evolução.

4.2.2 O Código-fonte Escolhido

Os trechos de código utilizados para realização do experimento foram retirados da REST-Fiddle, uma plataforma de gerenciamento de APIs para equipes que permite o projeto, desenvolvimento, testes e *releases* de APIs. Todo o código da plataforma está disponível no GitHub[1]. A Figura 4.5 apresenta a página inicial do site da plataforma[2].

[1] O código-fonte da plataforma RESTFiddle pode ser encontrado em: https://github.com/AnujaK/restfiddle
[2] O site da plataforma RESTFiddle pode ser acessado em: http://www.restfiddle.com/

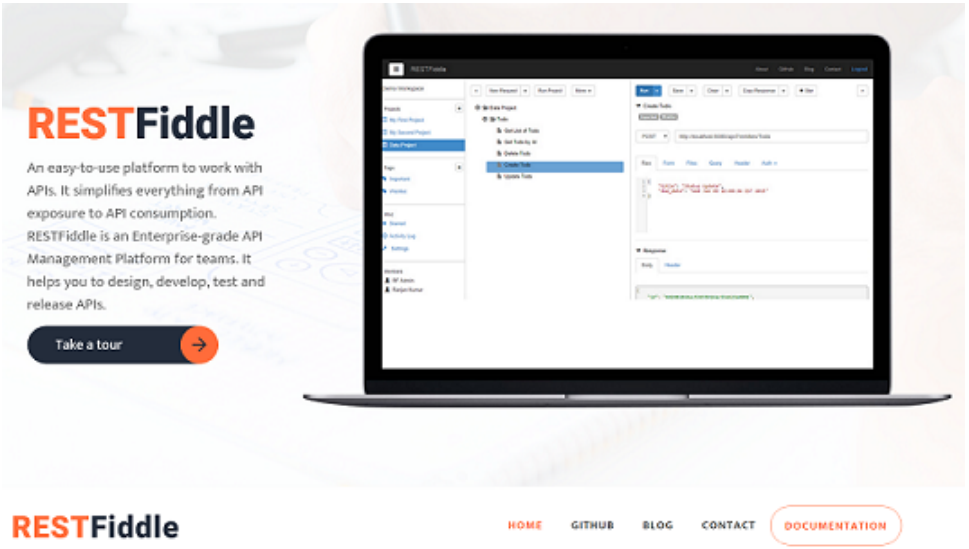


Figura 4.5: Página inicial do site da plataforma RESTFiddle

A Figura 4.6 apresenta o Código-fonte 1, o qual é a classe PasswordResetDTO, utilizada para manipulação de *passwords*.

```
/*
 * Copyright 2014 Ranjan Kumar
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

public class PasswordResetDTO {
    private String oldPassword;
    private String newPassword;
    private String retypedPassword;
    public String getOldPassword() {
        return oldPassword;
    }
    public void setOldPassword(String oldPassword) {
        this.oldPassword = oldPassword;
    }
    public String getNewPassword() {
        return newPassword;
    }
    public void setNewPassword(String newPassword) {
        this.newPassword = newPassword;
    }
    public String getRetypedPassword() {
        return retypedPassword;
    }
    public void setRetypedPassword(String retypedPassword) {
        this.retypedPassword = retypedPassword;
    }
}
```

Figura 4.6: Código-fonte 1 escolhido para o experimento

A Figura 4.7 contém o Código-fonte 2, o qual é a classe ErrorMessage, utilizada para alerta sobre erros ocorridos. É importante lembrar que o Código-fonte 2 foi modificado em relação ao

original através da adição do método "printReason()", o qual foi necessário para adequação do trecho ao propósito do experimento.

```
/**
 * Created by santoshm1 on 06/06/14.
 */
public class ErrorMessage {

    private String reason;

    public ErrorMessage(String reason) {
        this.reason = reason;
    }

    public ErrorMessage(Exception ex) {
        this.reason = ex.getMessage();
    }

    public String getReason() {
        return reason;
    }

    public void setReason(String reason) {
        this.reason = reason;
    }

    public void printReason(){
        System.out.println("Reason: " + this.reason);
    }

}
```

Figura 4.7: Código-fonte 2 escolhido para o experimento

Graças à baixa complexidade dos Códigos-fonte 1 e 2, pode-se assumir que cada método neles presente corresponde a uma *feature*. É importante ressaltar que o mesmo não acontece para todo tipo de código. Para trechos mais complexos, o nível de granularidade de *features* pode ser maior, permitindo a presença de uma série de funcionalidades dentro de um único método, por exemplo.

4.2.3 Considerações Finais

Embora os trechos de código-fonte utilizados possam ser considerados provenientes de um sistema legado, estes são pequenos e de baixa complexidade, o que não permite a clara visualização da influência disso sobre o resultado final do experimento. Mais detalhes sobre a estrutura utilizada para avaliação da ferramenta sobre o código-fonte durante o experimento serão apresentados no Capítulo 5.

Capítulo 5

Avaliação da Ferramenta

5.1 Procedimento para Realização do Experimento

Para a avaliação da ferramenta escolhida, foi utilizado o processo de experimentação apresentado por [Juristo e Moreno 2010], o qual indica que qualquer experimentação com qualquer nível de formalidade pode ser dividida em quatro atividades, cada uma gerando um produto de saída, como mostra a Figura 5.1.

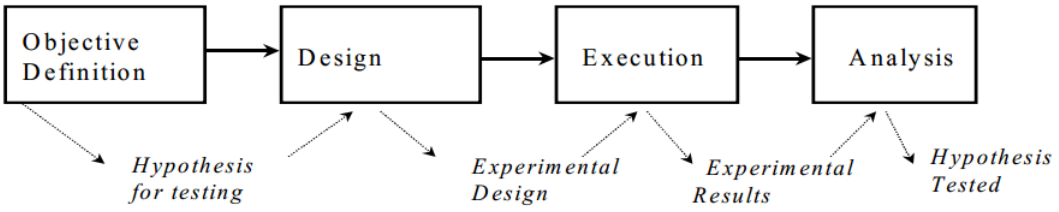


Figura 5.1: Etapas do processo de experimentação e seus produtos de saída [Juristo e Moreno 2010]

Durante a definição de objetivos, a hipótese geral é transformada em uma hipótese definida em termos de quais variáveis do fenômeno serão examinadas. Por exemplo, em um experimento onde se pretende examinar a qualidade de duas técnicas de testes para a detecção de um determinado tipo de erro, um experimentador pode definir uma hipótese quantificável como: A técnica A é capaz de detectar uma maior quantidade de erros de um tipo específico em relação à técnica B. Essa é uma hipótese quantificável no sentido de que pode ser mensurada.

A etapa de *design* envolve a realização de uma espécie de plano sobre quais experimentos devem ser realizados. O plano é formado pela determinação das condições sob as quais o experimento deve ser conduzido. Isso envolve a determinação de quais variáveis podem afetar

o experimento.

Na etapa de execução, experimentos elementares são executados assim como indicado pelo *design* selecionado. Após a realização dos experimentos, inicia-se a etapa de análise dos resultados, ou seja, a análise dos dados coletados durante os experimentos. Essa análise busca encontrar relações entre os resultados do estudo.

5.1.1 Terminologia

Para que se possa entender as etapas do processo de experimentação, é necessário que se tenha conhecimento sobre alguns termos básicos utilizados. Os conceitos utilizados no experimento aqui proposto são descritos a seguir [Juristo e Moreno 2010]:

- **Unidade experimental:** Os objetos nos quais o experimento é executado são chamados de unidades experimentais ou objetos experimentais. Por exemplo, pacientes são unidades experimentais em experimentos médicos, assim como cada pedaço de terra em experimentos de agricultura. Dependendo do objetivo do experimento, a unidade experimental na engenharia de *software* pode ser o projeto de *software* como um todo, ou qualquer produto gerado durante o processo. Em casos onde pretende-se comparar duas técnicas de testes, por exemplo, a unidade experimental seria o trecho de código no qual as técnicas são aplicadas.

- **Participantes do experimento:** As pessoas que aplicam os métodos ou técnicas às unidades experimentais são os participantes do experimento. No exemplo das técnicas de testes, os participantes seriam as pessoas que aplicam as técnicas. Ao contrário de outras disciplinas, os participantes do experimento na engenharia de *software* têm alto grau de influência sobre o resultado de experimentos. Desenvolvedores com diferentes níveis de conhecimento podem levar a resultados totalmente diferentes. Portanto, essa variável deve ser cuidadosamente selecionada durante o *design* do experimento.

- **Variável dependente ou de resposta:** O resultado de um experimento é conhecido como uma variável de resposta e deve ser quantitativo. Por exemplo, suponha que um pesquisador propõe uma nova técnica de estimativa de projetos e argumenta que ela fornece uma melhor estimativa em relação às técnicas existentes. O pesquisador deve então executar

um experimento com vários projetos, alguns usando a nova técnica e outros utilizando as já existentes. Uma possível variável de resposta nesses experimentos seria a precisão de estimativa.

- **Variável independente ou fator:** Cada característica de desenvolvimento de *software* a ser estudada, que afeta a variável de resposta, é chamada de fator e cada fator tem várias alternativas possíveis. A experimentação busca analisar a influência dessas alternativas sobre o valor da variável de resposta. Portanto, os fatores de um experimento são quaisquer características de projeto que são variadas intencionalmente durante a experimentação e que afetam os resultados do experimento.

- **Hipóteses estatísticas:** Quando se busca uma decisão estatística, é útil tentar construir hipóteses, que venham ou não a ser confirmadas posteriormente. Essas hipóteses são chamadas de hipóteses estatísticas.

  Muitas vezes, formula-se uma hipótese estatística com o único propósito de refutá-la. Assim, quando se busca decidir se uma moeda é falsa, por exemplo, cria-se a hipótese de que a moeda não é uma farsa. Da mesma forma, quando se procura decidir se uma alternativa é melhor que outra, utiliza-se a hipótese de que não há diferença entre as duas alternativas. Hipóteses dessa espécie são normalmente chamadas de hipóteses nulas. As hipóteses que de alguma forma diferem da descrição dada são chamadas de hipóteses alternativas.

5.2 Definição do Estudo

5.2.1 Introdução

O experimento envolveu uma avaliação da ferramenta FeatureIDE em relação ao apoio por ela fornecido nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos. Foram utilizados dezoito participantes no experimento, divididos em dois grupos com a mesma quantidade de integrantes. Em um primeiro momento, ambos os grupos realizaram as atividades do experimento utilizando apenas os recursos da linguagem na qual o código-fonte foi escrito, o qual foi diferente para cada equipe. Posteriormente,

os códigos foram trocados para que o experimento se repetisse, agora com apoio da ferramenta FeatureIDE.

5.2.2 Objeto de Estudo

Como objeto de estudo, foi utilizada a ferramenta escolhida através dos critérios definidos na Seção 4.1.2, FeatureIDE.

5.2.3 Objetivo Global

Avaliar a ferramenta FeatureIDE em relação ao apoio por ela fornecido para desenvolvedores de *software* legado nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, quando utilizando uma abordagem de LPS.

5.2.4 Contexto

Para que resultados mais efetivos sejam encontrados, o experimento deve ser executado em um contexto real, envolvendo uma empresa de *software* e desenvolvedores experientes. No entanto, isso muitas vezes não é viável, então uma alternativa mais barata é escolhida, como usar uma versão restrita do real ambiente ou estudantes como sujeitos experimentais. Tais abordagens são mais baratas e mais fáceis de controlar, mas raramente abordam problemas reais e são mais direcionadas para um determinado contexto. Assim, o contexto da experiência deve ser definido de acordo com as quatro dimensões seguintes [Wohlin et al. 2012]:

- ***Online* vs. *offline*:** O experimento foi executado presencialmente, portanto *offline*.
- **Estudantes vs. profissionais:** Os sujeitos do experimento são estudantes de graduação do curso de ciência da computação, sem conhecimento aprofundado sobre variabilidade e linhas de produtos de *software*.
- **Simulação vs. problema real:** O experimento aborda um problema real sob circunstâncias não reais. Por conta do tempo disponível e conhecimento limitado dos estudantes sobre o tema, foi utilizada uma versão simplificada do problema.
- **Específico vs. geral:** O experimento engloba um processo específico possível através da utilização da ferramenta FeatureIDE.

5.2.5 Questões

O objetivo do experimento foi mapeado para as duas questões (Q1 e Q2) apresentadas abaixo, as quais serão respondidas na etapa de análise dos resultados do experimento.

- **Questão 1 (Q1):** A ferramenta prove funcionalidades que satisfazem o usuário em suas necessidades declaradas e implícitas, no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos?

- **Questão 2 (Q2):** A ferramenta pode ser aprendida e operada por parte do usuário no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos?

5.2.6 Métricas

Para a definição das métricas associadas a cada questão, foi utilizada a norma NBR ISO/IEC 9126 [Guerra e Colombo 2009], uma norma ISO para qualidade de produtos de *software*. A Figura 5.2 apresenta o modelo para qualidade externa e interna contendo um conjunto de seis características de qualidade de produtos de *software* e suas respectivas sub-características. Em seguida, serão apresentadas as métricas definidas.

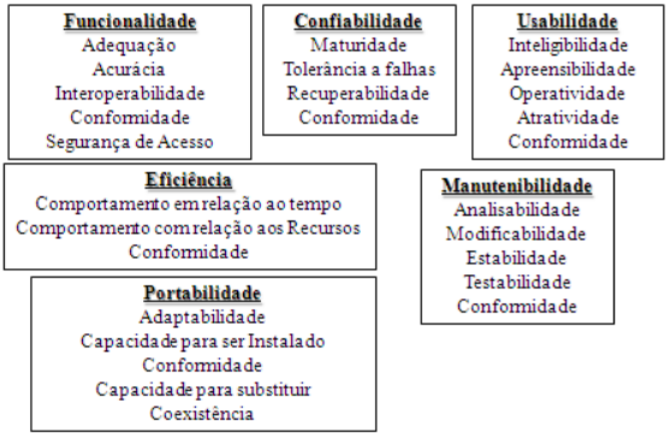


Figura 5.2: Características e sub-características de qualidade [Guerra e Colombo 2009]

- **M1:** Conhecimento do domínio dos códigos-fonte utilizados por parte dos participantes do experimento.
- **M2:** Dificuldade encontrada no processo de visualização e análise do modelo de *features*.
- **M3:** Dificuldade encontrada no processo de anotação do código-fonte.
- **M4:** Dificuldade encontrada no processo de configuração de produtos.
- **M5:** Número de *features* encontradas durante o experimento.
- **M6:** Corretude das anotações realizadas em código-fonte durante o experimento em relação ao que foi previsto.
- **M7:** Corretude da configuração de produtos realizada durante o experimento em relação ao que foi previsto.
- **M8:** Compreensibilidade das funcionalidades oferecidas pela ferramenta.
- **M9:** Adequação das funcionalidades oferecidas pela ferramenta em relação ao seu propósito.
- **M10:** Dificuldade encontrada para o processo de aprendizado da ferramenta.
- **M11:** Dificuldade encontrada para operação da ferramenta por parte do usuário.
- **M12:** Estabilidade da ferramenta em casos de erros.

A Tabela 4.1 apresenta a associação entre as questões definidas, as métricas utilizadas e o questionário de *feedback* utilizado no experimento. Nela, é possível visualizar também quais características e sub-características da NBR ISO/IEC 9126 foram levadas em consideração para coleta dos dados, as quais foram escolhidas com base nos dados necessários para resolução das questões Q1 e Q2. As células marcadas com o símbolo *, na última coluna da tabela, indicam que a coleta dos dados foi feita através de análise sobre os códigos utilizados pelos participantes do experimento.

Tabela 5.1: Associação entre as questões, métricas e o questionário de *feedback*

Questão	Característica	Sub-característica	Métrica	Questionário de *feedback*
Q1	Funcionalidade	Adequação	M1	O domínio dos códigos-fonte utilizados no experimento era conhecido.
			M2	A ferramenta tornou mais fácil o processo de visualização e análise do modelo de features.
			M3	A ferramenta tornou mais fácil o processo de anotação do código-fonte.
			M4	A ferramenta tornou mais fácil o processo de configuração de produtos.
		Acurácia	M5	Número de *features* encontradas. *
			M6	Análise da corretude das anotações realizadas em código-fonte em relação ao que foi previsto. *
			M7	Análise da corretude da configuração de produtos realizada durante o experimento em relação ao que foi previsto. *
Q2	Usabilidade	Inteligibilidade	M8	Foi fácil compreender as funcionalidades oferecidas pela ferramenta.
			M9	As funcionalidades disponíveis na ferramenta satisfazem o seu propósito.
			M10	O aprendizado da ferramenta foi fácil.
		Apreensibilidade	M11	A ferramenta facilitou a utilização (operação) por parte do usuário.
		Operacionalidade	M12	Em caso de erros, a ferramenta se manteve estável e não causou perdas.

5.2.7 Hipóteses

Para cada questão definida, foram criadas uma hipótese nula (H0) e uma alternativa (Ha), as quais serão aceitas ou rejeitadas no Capítulo 5, através da análise dos resultados do experimento.

- **Q1**
  - **H0:** O desempenho obtido nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos foi o mesmo com e sem a utilização da ferramenta, ou seja, a ferramenta FeatureIDE não fornece apoio significativo para as etapas mencionadas.
  - **Ha:** Há uma diferença significativa de desempenho dos participantes do experimento quando utilizando a ferramenta nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, ou seja, a ferramenta FeatureIDE fornece apoio significativo para as etapas mencionadas.

- **Q2**

  - **H0:** A ferramenta não pode ser completamente aprendida e operada sem grandes dificuldades por parte do usuário no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos.
  - **Ha:** A ferramenta pode ser aprendida e operada sem grandes dificuldades por parte do usuário no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos.

5.3 *Design*

O *design* do experimento é composto por um fator e dois tratamentos [Wohlin et al. 2012] e foi planejado visando compreender o significado das variáveis dependentes para cada tratamento. Mais detalhes a respeito do *design* serão apresentados a seguir.

5.3.1 Definição das Variáveis

- **Variável independente:** A variável independente ou fator que será variado intencionalmente durante o experimento buscando entender a sua influência nas variáveis de resposta é o método de implementação (com e sem o apoio da ferramenta FeatureIDE).
- **Variáveis dependentes:** Funcionalidade da ferramenta FeatureIDE, no que diz respeito à sua adequação e acurácia no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos, e usabilidade, a qual envolve inteligibilidade, apreensibilidade e operacionalidade.

5.3.2 Seleção dos Participantes do Experimento

Dezoito estudantes cursando a disciplina Processo de Engenharia de *Software* II do curso de ciência da computação da Universidade Estadual do Oeste do Paraná, divididos randomicamente em dois grupos de nove integrantes.

5.3.3 Estrutura do Experimento

A Tabela 4.2 apresenta a estrutura seguida durante a execução do experimento. Em um primeiro momento, houve a aplicação de um questionário de *background*, com o intuito de coletar a base de conhecimento de cada participante sobre o tema abordado. Na segunda etapa, ocorreu um treinamento sobre variabilidade e utilização da ferramenta FeatureIDE (*slides* de treinamento disponíveis no Apêndice A). Na terceira etapa, cada grupo formado realizou as atividades do experimento sobre um código-fonte exclusivo, sem apoio da ferramenta FeatureIDE. Na quarta etapa, os códigos utilizados foram trocados entre os grupos para que o experimento fosse então realizado com apoio da ferramenta. Por fim, na quinta etapa, foi aplicado um questionário de *feedback*, com o intuito de coletar dados a respeito da utilização da FeatureIDE. Os questionários de *background* e *feedback* estão disponíves nos Apêndices B e C, respectivamente.

Tabela 5.2: Estrutura do experimento

	Grupo 1	Grupo 2		Tempo
	Questionário de *background*		Primeira etapa	10min
	Treinamento: Apresentação de conceitos básicos de variabilidade e da ferramenta FeatureIDE		Segunda etapa	60min
Sem apoio da ferramenta FeatureIDE	Código-fonte 1	Código-fonte 2	Terceira etapa	70min
Com apoio da ferramenta FeatureIDE	Código-fonte 2	Código-fonte 1	Quarta etapa	70min
	Questionário de *feedback*		Quinta etapa	10min

Nas etapas três e quatro, os participantes receberam as árvores de *features* já prontas, as quais foram entregues via imagem, no caso da etapa sem o apoio da ferramenta, e via projeto da FeatureIDE, na etapa com apoio da mesma. Além disso, houve a indicação das características de cada produto a ser gerado (*features* a serem incluídas no produto final), como mostra a Figura 5.3. Neste caso, como sugerem as *features* da árvore, a mesma está relacionada ao Código-fonte 2, apresentado na Seção 4.2.2.

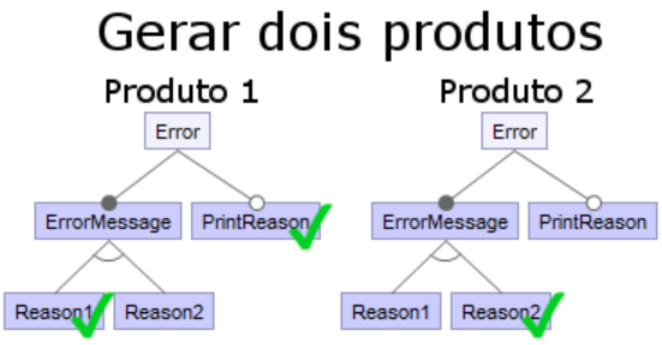


Figura 5.3: Produtos a serem gerados em uma das etapas do experimento

5.3.4 Instrumentação

O *background* e experiência dos participantes do experimento foram coletados através do questionário disponível no Apêndice B. Da mesma forma, para coleta das opiniões dos participantes após a execução do experimento, foi utilizado um questionário de *feedback*, disponível no Apêndice C.

Para execução do experimento, todos os participantes utilizaram a mesma versão do ambiente de desenvolvimento Eclipse com a adição do *plugin* da ferramenta FeatureIDE.

5.3.5 Mecanismos de Análise

Para interpretação dos dados coletados na execução do experimento, foram utilizados alguns mecanismos para análise de dados. Estes mecanismos serão descritos a seguir.

- **Escala Likert:** A escala Likert ou escala de Likert é um tipo de escala de resposta psicométrica usada habitualmente em questionários, e é a escala mais usada em pesquisas de opinião. Ao responderem a um questionário baseado nesta escala, os perguntados especificam seu nível de concordância com uma afirmação [Likert 1932]. Para análise dos dados, a estrutura da escala é geralmente utilizada da seguinte forma:
  - Discordo totalmente
  - Discordo
  - Não concordo nem discordo
  - De acordo

- Totalmente de acordo

A Escala Likert foi utilizada nos questionários de *background* e *feedback* para coleta dos conhecimentos e das opiniões dos participantes do experimento em relação à utilização da ferramenta FeatureIDE.

- **Teste-t para duas amostras:** O teste-t é um teste paramétrico utilizado para comparar duas amostras independentes. O *design* apropriado para sua utilização é o que apresenta um fator e dois tratamentos [Wohlin et al. 2012]. De forma geral, o teste-t para duas amostras consiste de um teste de hipótese que usa conceitos estatísticos para rejeitar ou não uma hipótese nula em casos onde se deseja comparar duas amostras independentes.

  O teste-t para duas amostras foi utilizado para avaliar a influência dos trechos de código-fonte utilizados no resultado final do experimento.

Capítulo 6

Resultados e Análise dos Dados

Os resultados do experimento foram agrupados para cada afirmação baseada na Escala de Likert presente no questionário de *feedback*. Abaixo serão apresentados os dados percentuais sobre cada afirmação e a interpretação dos dados. Em seguida, as hipóteses para cada uma das questões anteriormente definidas (Q1 e Q2) serão testadas.

6.1 Respostas e Interpretação dos Dados

1. A ferramenta tornou mais fácil o processo de visualização e análise do modelo de *features*.

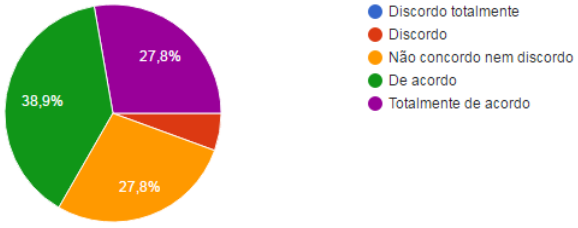


Figura 6.1: Respostas para a questão 5 do questionário de *feedback*

Analisando o gráfico da Figura 6.1, é possível notar que a maioria dos participantes do experimento (66,7%) esteve de acordo ou totalmente de acordo que a ferramenta torna mais fácil o processo de visualização e análise do modelo de *features*. Porém, um dos dezoito participantes (5,5%) discordou da mesma afirmação. Este mesmo indivíduo relatou ter tido problemas na execução da ferramenta. Com a posterior análise do código-fonte por ele alterado, pôde-se notar que os problemas se originaram de erros no processo de anotação do código-fonte - as*features* indicadas nas anotações não estavam presentes na

árvore de *features*, ocasionando a não inclusão das mesmas no produto final, como mostra a Figura 6.2. Assim, o experimento sugere que a ferramenta FeatureIDE facilita a tarefa de visualização e análise do modelo de *features*.

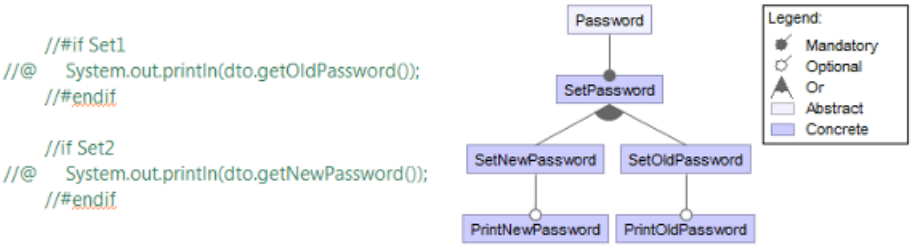


Figura 6.2: Erro de anotação do código-fonte na ferramenta FeatureIDE

2. A ferramenta tornou mais fácil o processo de anotação do código-fonte.

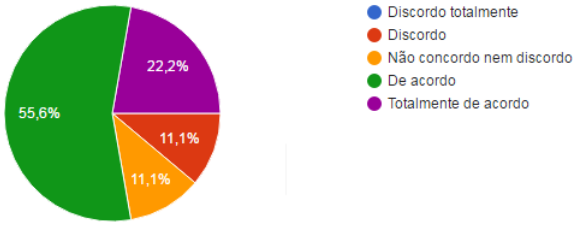


Figura 6.3: Respostas para a questão 6 do questionário de *feedback*

Através do gráfico apresentado na Figura 6.3, pode-se perceber que 66,7% dos participantes do experimento concordam, de alguma forma, que a ferramenta FeatureIDE torna mais fácil o processo de anotação do código-fonte. Embora 11.1% esteja em desacordo sobre isso, é importante mencionar que as anotações do código-fonte por eles realizadas estavam incompletas - alguns trechos de código pertencentes a *features* da árvore de *features* não estavam circundados. Ademais, nos trechos anotados, foi possível perceber que a sintaxe das anotações estava correta. Assim, uma hipótese que explicaria os trechos não circundados é a dificuldade no processo de visualização e análise do modelo de *features*, mas, como pode ser visto no item anterior, a mesma hipótese foi desconsiderada. Assim, o experimento sugere que a FeatureIDE facilita o processo de anotação do código-fonte.

3. A ferramenta tornou mais fácil o processo de configuração de produtos.

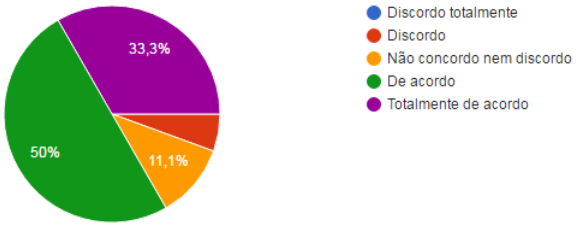


Figura 6.4: Respostas para a questão 7 do questionário de *feedback*

A maioria dos participantes também esteve de acordo ou totalmente de acordo que a ferramenta FeatureIDE torna mais fácil o processo de configuração de produtos, como mostra a Figura 6.4. Apenas um dos dezoito participantes esteve em desacordo sobre a afirmação. Com isso, o experimento sugere que a FeatureIDE, em linhas gerais, apoia o processo de configuração de produtos.

4. Foi fácil compreender as funcionalidades oferecidas pela ferramenta.

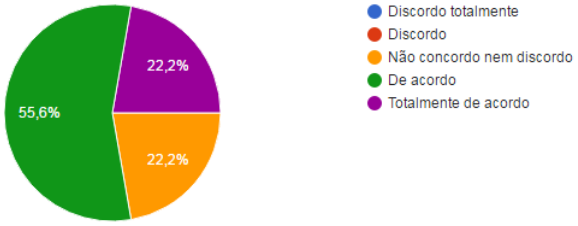


Figura 6.5: Respostas para a questão 8 do questionário de *feedback*

Neste caso, não houve participantes que discordaram da afirmação. Como pode ser visto na Figura 6.5, 55,6% dos participantes esteve de acordo que as funcionalidades oferecidas pela ferramenta são de fácil compreensão. Além disso, 22,2% deles esteve totalmente de acordo com a mesma afirmação. Esta parcela aplica-se também para os indivíduos que não concordaram nem discordaram sobre a afirmação. Assim, o experimento sugere que a ferramenta FeatureIDE tem um conjunto de funcionalidades bem especificado e com um propósito bem definido.

5. As funcionalidades disponíveis na ferramenta satisfizeram o seu propósito.

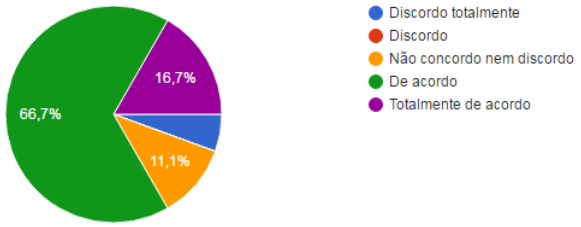


Figura 6.6: Respostas para a questão 9 do questionário de *feedback*

Pode-se notar, através da visualização da Figura 6.6, que a grande maioria dos participantes do experimento esteve de acordo, de alguma forma, que as funcionalidades disponíveis na ferramenta FeatureIDE satisfazem o seu propósito. No entanto, um dos dezoito participantes discordou totalmente sobre a mesma afirmação, mantendo-se destoante do restante das respostas para esse caso. Contudo, o código-fonte alterado pelo mesmo indivíduo durante o experimento apresentou erros de anotação e configuração de produtos, o que pode justificar a resposta divergente. Dessa maneira, o experimento sugere que a FeatureIDE possui os módulos necessários para o cumprimento de seu propósito.

6. O aprendizado da ferramenta foi fácil.

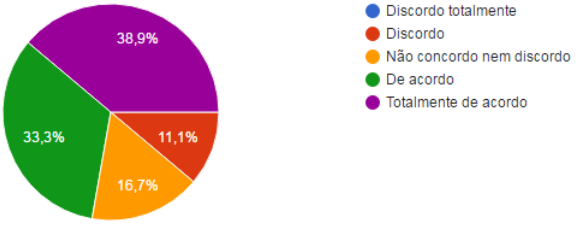


Figura 6.7: Respostas para a questão 10 do questionário de *feedback*

Olhando para o gráfico da Figura 6.7, nota-se que a maioria dos participantes do experimento concordou, de alguma forma, que o aprendizado da ferramenta FeatureIDE é fácil - 33,3% estiveram de acordo e 38,9% estiveram totalmente de acordo. Três dos dezoito participantes (16,7%) não concordaram nem discordam sobre a afirmação e 11,1% deles discordaram sobre ela. Embora o treinamento dos participantes tenha sido planejado e

executado de forma a cobrir todas as tarefas necessárias na ferramenta para cumprimento dos objetivos do experimento, uma possibilidade para a discordância é o não entendimento do processo durante o treinamento por parte desses indivíduos. Assim, o experimento sugere que, para o processo nele coberto, a ferramenta FeatureIDE é de fácil aprendizado.

7. A ferramenta facilitou a utilização (operação) por parte do usuário.

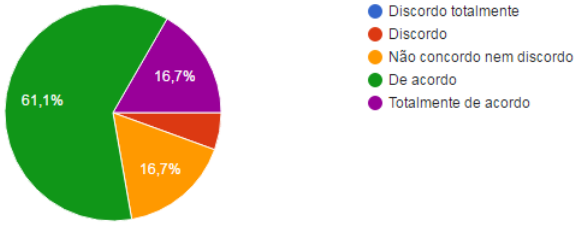


Figura 6.8: Respostas para a questão 11 do questionário de *feedback*

Analisando o gráfico da Figura 6.8, é possível notar que 78,4% dos participantes do experimento esteve de acordo ou totalmente de acordo que a ferramenta facilita a utilização (operação) por parte do usuário. Apenas um dos dezoito participantes discordou sobre a afirmação. Contudo, através da análise das respostas do questionário de *background* fornecidas pelo mesmo indivíduo, pôde-se perceber que este não possuía conhecimento sobre o ambiente de desenvolvimento Eclipse, o que certamente influencia na operação da ferramenta, considerando que ela foi executada sobre o mesmo ambiente. Assim, o experimento sugere que a FeatureIDE é planejada de forma a minimizar dificuldades de operação por parte do usuário.

8. Em caso de erros, a ferramenta se manteve estável e não causou perdas.

   Como pode ser visto na Figura 6.9, não houve participantes que discordaram que a ferramenta FeatureIDE, em caso de erros, se mantém estável e não causa perdas. É possível notar que uma parcela de 50% dos participantes não concordou nem discordou da afirmação. Uma hipótese para explicar o ocorrido é a ausência de situações críticas durante o experimento para esses indivíduos. Por fim, a outra metade dos participantes, a qual concordou com a afirmação, indica que a FeatureIDE consegue lidar com erros e continuar

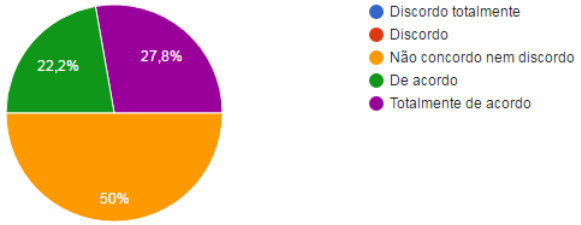


Figura 6.9: Respostas para a questão 12 do questionário de *feedback*

funcionando sem causar perdas de progresso.

6.2 Teste das Hipóteses

A seguir, as hipóteses anteriormente definidas sobre as questões Q1 e Q2 serão testadas com base nas informações obtidas através da análise e interpretação dos dados apresentados na seção anterior.

- **Q1**
  - H0: O desempenho obtido nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos foi o mesmo com e sem a utilização da ferramenta, ou seja, a ferramenta FeatureIDE não fornece apoio significativo para as etapas mencionadas.
  - Ha: Há uma diferença significativa de desempenho dos participantes do experimento quando utilizando a ferramenta nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, ou seja, a ferramenta FeatureIDE fornece apoio significativo para as etapas mencionadas.

Analisando as respostas coletadas, é possível perceber que a ferramenta FeatureIDE satisfaz, de fato, as necessidades do usuário no processo de visualização e análise da árvore de *features*, anotação do código e configuração de produtos. Contudo, o tempo necessário para realização da atividade na etapa com apoio da ferramenta foi, em média, maior (7,4 minutos na etapa sem o apoio da ferramenta, contra 8,16 minutos na etapa com apoio da mesma). Em contraste, houve vários relatos de participantes do experimento contestando

o maior tempo proveniente do uso da ferramenta. Por exemplo: "O uso da ferramenta poderia ser realmente medido em testes com projetos maiores. Não houve real impacto no tempo devido a simplicidade do projeto teste. Suponho, porém, que o resultado possa ser positivo em testes complexos". De fato, os códigos utilizados no experimento eram relativamente curtos e simples, o que certamente prejudica a visualização de uma diferença significativa de tempo entre as etapas com e sem apoio da ferramenta FeatureIDE.

Além disso, é importante lembrar que a ferramenta demandou tempo para adaptação por parte dos usuários. Um dos participantes relatou: "Meu único adendo vai pelo fato de que como qualquer ferramenta ela possui um período de adaptação (ficar lembrando de colocar as *tags* em cada *feature* variável do *software*)". Assim, utilizando as informações obtidas e com o apoio dos relatos dos participantes do experimento, pode-se rejeitar a hipótese H0.

- **Q2**

  - H0: A ferramenta não pode ser aprendida e operada sem grandes dificuldades por parte do usuário no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos.
  - Ha: A ferramenta pode ser aprendida e operada sem grandes dificuldades por parte do usuário no processo de visualização/análise da árvore de *features*, anotação do código e configuração de produtos.

Através da análise dos dados apresentados na seção anterior associados à questão Q2, pode-se concluir que a ferramenta pode ser aprendida e operada sem grandes dificuldades. Como mostrado anteriormente, alguns indivíduos tiveram problemas no processo de anotação do código-fonte. Entretanto, erros desta espécie eram esperados visto que os participantes não usaram anotações para lembrar de detalhes a respeito do processo, incluindo a sintaxe correta de anotação. Com essas informações em mãos, é possível rejeitar a hipótese H0.

6.3 Ameaças à Validade do Experimento

Devido à natureza da experimentação na engenharia de *software*, as ameaças à validade dos resultados obtidos devem ser apontadas. Para minimizar as ameaças, algumas ações foram tomadas durante o projeto do experimento. Uma delas foi o cruzamento dos códigos-fonte utilizados entre os dois grupos. Com isso, cada grupo realizou as etapas sem e com o apoio da ferramenta sobre trechos de código diferentes. A decisão foi tomada para evitar que as características de cada código influenciassem no resultado final do experimento.

A Tabela 5.1 apresenta um teste-t realizado para comparação das opiniões entre os dois grupos para a seguinte afirmação Likert presente no questionário de *feedback*: "A ferramenta tornou mais fácil o processo de visualização e análise do modelo de features". Para análise dos dados, as respostas inseridas através da escala Likert foram codificadas da seguinte forma: Discordo totalmente = 1; Discordo = 2; Não concordo nem discordo = 3; De acordo = 4; Totalmente de acordo = 5.

Tabela 6.1: Teste-t com duas amostras (grupos 1 e 2) presumindo variâncias equivalentes

	Grupo 1	***Grupo 2***
Média	4,111111111	3,888888889
Variância	0,861111111	0,861111111
Observações	9	9
Variância agrupada	0,861111111	
Hipótese da diferença de média	0	
gl	16	
Stat t	0,508000508	
P(T<=t) uni-caudal	0,309192255	
t crítico uni-caudal	1,745883676	
P(T<=t) bi-caudal	0,61838451	
t crítico bi-caudal	2,119905299	

O teste executado é bi-caudal, ou seja, considera as duas extremidades da distribuição, e a hipótese nula definida foi a equivalência entre as variâncias dos dois grupos. O valor de α (nível de significância) utilizado foi 0,05 (5%). Como mostra a Tabela 5.1, o valor de P para o teste bi-caudal é superior ao valor de α, o que indica a aceitação da hipótese nula e de que as características dos trechos de código utilizados não interferiram no resultado do experimento.

O mesmo aconteceu para todas as outras questões do questionário de *feedback*. Mesmo assim, existem algumas ameaças à validade do experimento que não puderam ser evitadas, as

quais serão discutidas a seguir.

6.3.1 Validade de Construção

A validade de construção considera os relacionamentos entre a teoria e a observação, ou seja, se o tratamento reflete bem a causa e se o resultado reflete bem o efeito [Travassos, Gurov e Amaral 2002].

Como pode ser visto na estrutura do experimento apresentada na Tabela 5.2, os dois grupos cruzaram os códigos-fonte utilizados, mas ambos realizaram primeiramente a etapa sem o apoio da ferramenta. Assim, é importante salientar que o cruzamento das etapas, com um dos grupos iniciando o experimento com o apoio da FeatureIDE e o outro iniciando sem o mesmo, poderia modificar o resultado final.

6.3.2 Validade Interna

A validade interna, para [Travassos, Gurov e Amaral 2002], "define se o relacionamento observado entre o tratamento e o resultado é causal, e não é resultado da influência de outro fator - não controlado ou medido".

Embora os participantes tenham sido divididos em grupos com o mesmo número de integrantes, a seleção, por ser aleatória, não levou em conta questões de *background* dos participantes. Assim, é possível que tenha existido assimetria entre os dois grupos, em relação ao conhecimento possuído, fato que pode influenciar o resultado final do experimento.

Além disso, vale lembrar que o projeto inicial do experimento contava com a participação de vinte participantes e não dezoito, como realmente aconteceu. Dois dos participantes compareceram na etapa de treinamento, mas não estiveram presentes na execução do experimento. As características específicas desses indivíduos poderiam modificar o resultado final.

Por fim, a execução do experimento foi planejada de forma a minimizar as interações com os participantes do experimento. Contudo, houve casos onde a interação foi necessária. Alguns indivíduos, durante a execução do experimento, abriram e utilizaram outra versão do ambiente de desenvolvimento Eclipse disponível nos computadores usados, mesmo após orientações sobre qual versão utilizar. Um dos participantes relatou: "O problema foi que abri o outro Eclipse instalado no computador, e ele não tinha a ferramenta instalada. Então acabei demorando por

causa disso". Essas situações, além de terem exigido interações, interferiram no tempo final na etapa com o apoio da ferramenta FeatureIDE.

6.3.3 Validade Externa

A validade externa define condições que limitam a habilidade de generalizar os resultados de um experimento para a prática industrial [Travassos, Gurov e Amaral 2002].

Por conta das dificuldades encontradas no projeto do experimento, os participantes selecionados não representam quantitativamente e qualitativamente os reais usuários da ferramenta, os quais seriam desenvolvedores experientes trabalhando em empresas de *software*. Além disso, os trechos de código utilizados, apesar de provenientes de *software* legado, são pequenos e de baixa complexidade, escolhidos assim por conta do tempo disponível e pelo conhecimento restrito dos participantes sobre o domínio dos códigos. Assim, o resultado final do experimento pode apresentar diferenças significativas se aplicado a um contexto real e não simulado.

6.4 Considerações Finais

Como visto, os resultados do experimento indicam que a ferramenta FeatureIDE possui a capacidade de apoiar desenvolvedores de *software* nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos, quando utilizando uma abordagem de LPS. Assim, sugere-se que a FeatureIDE possa ser utilizada para ajudar empresas de tecnologia no processo de migração de sistemas legados para uma linha de produtos de *software*.

Capítulo 7

Conclusão e Trabalhos Futuros

Este trabalho apresentou um experimento para avaliação da ferramenta FeatureIDE em relação ao seu apoio fornecido nas etapas de visualização e análise da árvore de *features*, anotação do código-fonte e configuração de produtos. Como visto no Capítulo 6, as respostas coletadas com o questionário de *feedback* foram positivas em relação ao apoio da ferramenta nas etapas mencionadas. O tempo necessário para a conclusão do experimento na etapa com a utilização da ferramenta foi maior, como visto no Capítulo 6.

Contudo, a complexidade e tamanho dos códigos-fonte utilizados não permitem a visualização de uma clara diferença de desempenho em relação ao tempo. Além do período necessário para adaptação à ferramenta por parte dos participantes, foram necessárias interações durante o experimento para solução de problemas encontrados.

Assim, os resultados sugerem que a ferramenta FeatureIDE é capaz de apoiar empresas de tecnologia no processo de migração de sistemas legados para uma linha de produtos de *software*. Entretanto, é importante lembrar que o experimento abordou um problema real, porém em um ambiente simulado.

7.1 Trabalhos Futuros

Como possíveis trabalhos futuros, pode-se apontar a realização do experimento em um contexto real, envolvendo desenvolvedores experientes trabalhando em empresas de *software*, e a utilização de trechos de código-fonte maiores e mais complexos no experimento.

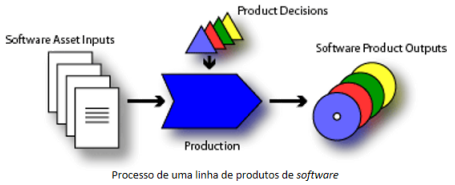
Apêndice A

Slides de Treinamento

Figura A.1: Treinamento - Slide 1

Linhas de Produtos de *Software*

- Conjunto de sistemas que utilizam *software* intensivamente, compartilhando um conjunto de características comuns e gerenciadas, que satisfazem as necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de um conjunto comum de ativos principais e de uma forma preestabelecida.



Processo de uma linha de produtos de *software*

Figura A.2: Treinamento - Slide 2

Conceitos relacionados

- **Variabilidade**
  - Habilidade de se derivar vários produtos a partir de um conjunto comum de artefatos.
- **Variante**
  - Termo utilizado para se referir a uma variante particular de uma *feature* variante.
- **Ponto de variação**
  - Determinados locais de um sistema de *software* onde são feitas escolhas sobre qual variante utilizar.

Figura A.3: Treinamento - Slide 3

Feature

- Característica ou um comportamento visível pelo usuário final de um sistema de *software*.

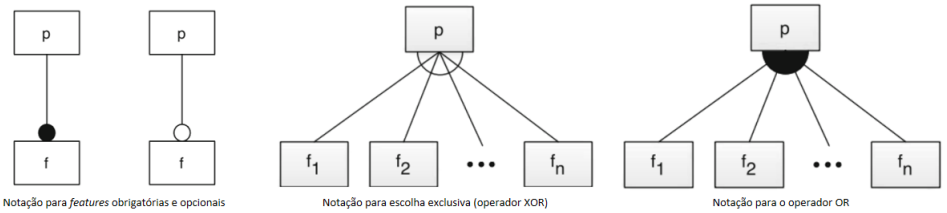
- São usadas para distinguir produtos
  - "O aplicativo a ser desenvolvido deve funcionar tanto em Android quanto em iOS."
  - "Ambas as aplicações financeiras suportam transações internacionais."
  - "O reprodutor de vídeos A suporta o formato .MP4, o reprodutor de vídeos B não."

Figura A.4: Treinamento - Slide 4

Diagrama de *features*

- Notação visual utilizada para especificação de um modelo de *features*. Consiste de uma árvore onde os nós representam os nomes de cada *feature*.

- **Notação**



Notação para *features* obrigatórias e opcionais

Notação para escolha exclusiva (operador XOR)

Notação para o operador OR

Figura A.5: Treinamento - Slide 5

Diagrama de *features*

- **Exemplo**

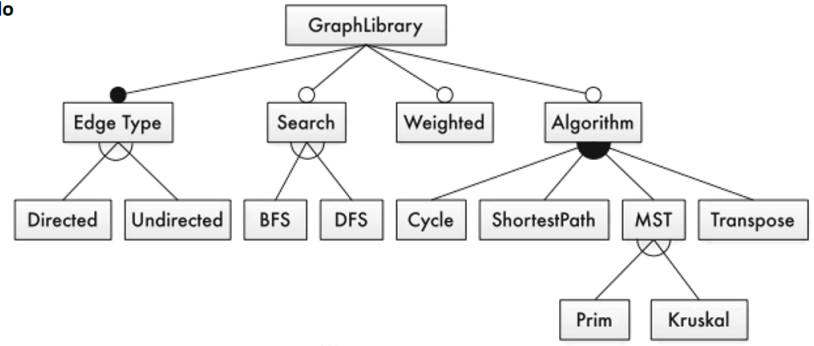
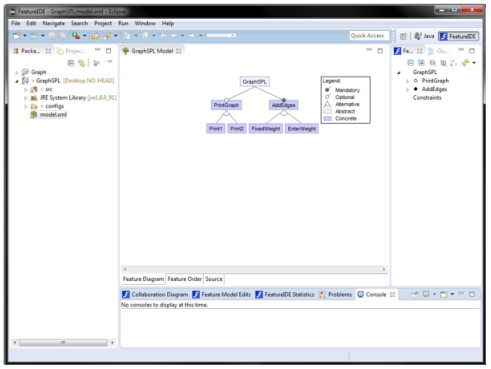


Diagrama de *features* para a implementação de um grafo

Figura A.6: Treinamento - Slide 6

FeatureIDE

- *Plugin* disponível no ambiente de desenvolvimento Eclipse.



Aparência da ferramenta FeatureIDE

Figura A.7: Treinamento - Slide 7

FeatureIDE

- Diagrama de *features*

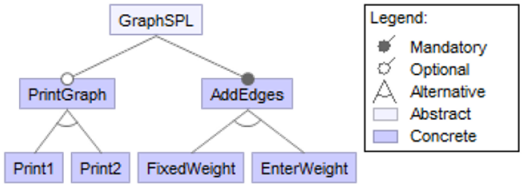


Diagrama de *features* na ferramenta FeatureIDE

Figura A.8: Treinamento - Slide 8

FeatureIDE

- Anotação do código-fonte

```
//#if Feature1
        Trecho de código associado à Feature1
//#endif
```

Figura A.9: Treinamento - Slide 9

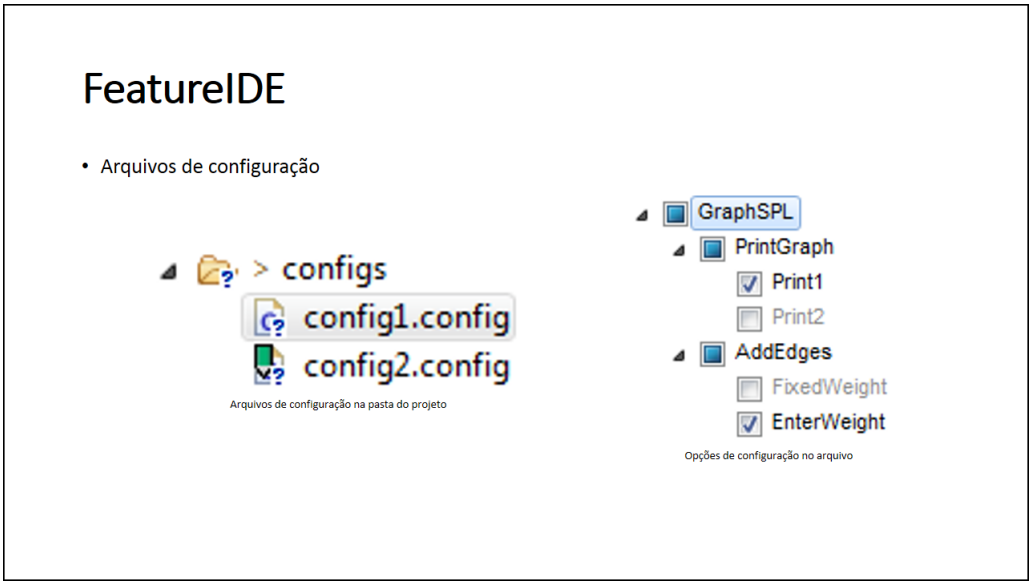


Figura A.10: Treinamento - Slide 10

Figura A.11: Treinamento - Slide 11

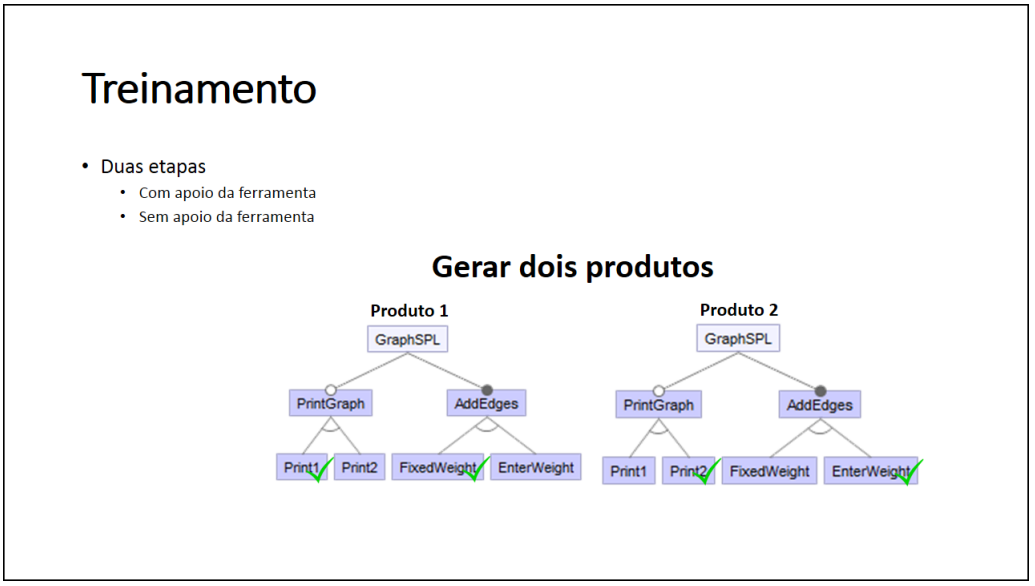


Figura A.12: Treinamento - Slide 12

Apêndice B

Questionário de *Background*

Este questionário tem por objetivo a coleta de informações a respeito dos conhecimentos e experiências dos participantes do experimento. Por favor, preencha-o de maneira precisa, de acordo com seu nível de conhecimento.

1. Qual seu nome? _____

2. Qual o seu conhecimento sobre a linguagem Java?

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

3. Qual o seu conhecimento sobre o ambiente de desenvolvimento Eclipse?

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

4. Qual o seu conhecimento em pré-processamento de código-fonte?

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

5. Qual o seu conhecimento em anotação de código-fonte em Java?

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

6. Qual o seu conhecimento em modelagem de *features*?

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

7. Qual o seu conhecimento em variabilidade de *software*?

   * Considere variabilidade como a habilidade de se derivar vários produtos a partir de um conjunto comum de artefatos.

   ( ) Nenhum

   ( ) Teórico

   ( ) Prático, na universidade

   ( ) Prático, na indústria

8. Qual o seu conhecimento em reuso de *software*?

    * Considere reuso de *software* como o uso de conceitos, produtos ou soluções previamente elaboradas ou adquiridas para criação de um novo *software*, visando aprimorar a qualidade e a produtividade.

    ( ) Nenhum

    ( ) Teórico

    ( ) Prático, na universidade

    ( ) Prático, na indústria

9. Qual o seu conhecimento em linhas de produtos de *software*?

    ( ) Nenhum

    ( ) Teórico

    ( ) Prático, na universidade

    ( ) Prático, na indústria

Apêndice C

Questionário de *Feedback*

Este questionário tem por objetivo a coleta de informações e *feedback* a respeito do processo de experimentação. Por favor, preencha-o de maneira precisa, de acordo com o que você presenciou durante as duas etapas do experimento.

1. Qual seu nome? _____
2. Quanto tempo (minutos) levou a etapa sem o apoio da ferramenta? _____
3. Quanto tempo (minutos) levou a etapa com o apoio da ferramenta? _____
4. O domínio dos códigos-fonte utilizados no experimento era conhecido.
   - ( ) Discordo totalmente
   - ( ) Discordo
   - ( ) Não concordo nem discordo
   - ( ) De acordo
   - ( ) Totalmente de acordo

5. A ferramenta tornou mais fácil o processo de visualização e análise do modelo de *features*.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

6. A ferramenta tornou mais fácil o processo de anotação do código-fonte.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

7. A ferramenta tornou mais fácil o processo de configuração de produtos.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

8. Foi fácil compreender as funcionalidades oferecidas pela ferramenta.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

9. As funcionalidades disponíveis na ferramenta satisfizeram o seu propósito.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

10. O aprendizado da ferramenta foi fácil.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

11. A ferramenta facilitou a utilização (operação) por parte do usuário.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

12. Em caso de erros, a ferramenta se manteve estável e não causou perdas.

   ( ) Discordo totalmente

   ( ) Discordo

   ( ) Não concordo nem discordo

   ( ) De acordo

   ( ) Totalmente de acordo

13. Escreva aqui quaisquer observações e/ou críticas relacionadas ao experimento e à ferramenta FeatureIDE (opcional).

_____

_____

_____

_____

_____

Referências Bibliográficas

[Apel et al. 2013]APEL, S. et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. 1. ed. Berlin/Heidelberg: Springer-Verlag, 2013.

[Apel, Kästner e Christian 2009]APEL, S.; KäSTNER, C.; CHRISTIAN, L. Featurehouse: Language-independent, automated software composition. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. Los Alamitos, CA: [s.n.], 2009. p. 221–231.

[Apel, Leich e Marnitz 2005]APEL, S.; LEICH, T.; MARNITZ, L. Proceedings of the 2005 oopsla workshop on eclipse technology exchange. In: 1. ed. New York, NY, USA: ACM, 2005. cap. Tool Support for Feature-oriented Software Development: FeatureIDE: an Eclipse-based Approach, p. 55–59.

[Apel et al. 2005]APEL, S. et al. Generative programming and component engineering, 4th international conference, GPCE 2005, tallinn, estonia, september 29 - october 1, 2005, proceedings. In: 1. ed. Berlin/Heidelberg: Springer-Verlag Berlin Heidelberg, 2005. cap. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming, p. 125–140.

[Appeltauer et al. 2011]APPELTAUER, M. et al. Information and media technologies. In: 1. ed. [S.l.]: Information and Media Technologies Editorial Board, 2011. cap. ContextJ: Context-oriented Programming with Java, p. 399–419.

[Bastos et al. 2011]BASTOS, J. F. et al. Evaluation assessment in software engineering (ease 2011), 15th annual conference on. In: 1. ed. Durham, UK, UK: IET, 2011. cap. Adopting software product lines: A systematic mapping study, p. 11–20.

[Bennett 1995]BENNETT, K. Legacy systems: coping with success. *IEEE Software*, Durham, UK, v. 12, n. 1, p. 19–23, Jan 1995.

[Brodie e Stonebraker 1995]BRODIE, M. L.; STONEBRAKER, M. *Migrating legacy systems: Gateways, interfaces the incremental approach*. 1. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995.

[Brownsword e Clements 1996]BROWNSWORD, L.; CLEMENTS, P. *A Case Study in Successful Product Line Development*. Pittsburgh, PA, Outubro 1996.

[Clements e Northrop 2001]CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. 1. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[Cohen 2002]COHEN, S. *Product Line State of the Practice Report*. Software Engineering Institute, Setembro 2002.

[Guerra e Colombo 2009]GUERRA, A. C.; COLOMBO, R. M. T. *Qualidade de Produto de Software*. 1. ed. Brasília, BR: Ministério da Ciência, Tecnologia e Inovação, 2009.

[Hirschfeld, Costanza e Haupt 2008]HIRSCHFELD, R.; COSTANZA, P.; HAUPT, M. Generative and transformational techniques in software engineering ii: International summer school, gttse 2007, braga, portugal, july 2-7, 2007. revised papers. In: 1. ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. cap. An Introduction to Context-Oriented Programming with ContextS, p. 396–407.

[Juristo e Moreno 2010]JURISTO, N.; MORENO, A. M. *Basics of Software Engineering Experimentation*. 1. ed. New York: Springer Publishing Company, Incorporated, 2010.

[Kästner, Apel e Kuhlemann 2008]KäSTNER, C.; APEL, S.; KUHLEMANN, M. Proceedings of the 30th international conference on software engineering. In: 1. ed. New York, NY, USA: ACM, 2008. cap. Granularity in Software Product Lines, p. 311–320.

[Krueger 2002]KRUEGER, C. W. Revised papers from the 4th international workshop on software product-family engineering. In: 1. ed. London, UK, UK: Springer-Verlag, 2002. cap. Easing the Transition to Software Mass Customization, p. 282–293.

[Likert 1932]LIKERT, R. A technique for the measurement of attitudes. *Archives of Psychology*, New York, v. 22, n. 140, p. 1–55, 1932.

[Svahnberg, Gurp e Bosch 2005]SVAHNBERG, M.; GURP, J. van; BOSCH, J. A taxonomy of variability realization techniques: Research articles. In: *Softw. Pract. Exper.* New York, NY, USA: [s.n.], 2005. p. 705–754.

[Travassos, Gurov e Amaral 2002]TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. do. *Introdução à Engenharia de Software Experimental*. Rio de Janeiro, BR, Outubro 2002.

[Wohlin et al. 2012]WOHLIN, C. et al. *Experimentation in Software Engineering*. 2012. ed. Norwell, MA, USA: Springer, 2012.