UFRB - Universidade Federal do Recôncavo da Bahia
CETEC - Centro de Ciências Exatas e Tecnológicas
Bacharelado de Engenharia da Computação

INVESTIGAÇÃO DE TÉCNICAS DE IMPLEMENTAÇÃO DE VARIABILIDADE PARA *FRAMEWORKS JAVASCRIPT* MODERNOS

TAIRONE CONCEIÇÃO DIAS

CRUZ DAS ALMAS
2019

TAIRONE CONCEIÇÃO DIAS

INVESTIGAÇÃO DE TÉCNICAS DE IMPLEMENTAÇÃO DE VARIABILIDADE PARA *FRAMEWORKS JAVASCRIPT* MODERNOS

Trabalho de conclusão de curso apresentado à Universidade Federal do Recôncavo da Bahia como parte dos requisitos para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Tassio Ferreira Vale

Cruz das Almas
2019

UNIVERSIDADE FEDERAL DO RECÔNCAVO DA BAHIA

CETEC – CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

INVESTIGAÇÃO DE TÉCNICAS DE IMPLEMENTAÇÃO DE VARIABILIDADE PARA *FRAMEWORKS JAVASCRIPT* MODERNOS

Aprovada em: 05/07/2019

BANCA EXAMINADORA:

ASS _____
Presidente: Dr. Tassio Ferreira Vale

ASS _____
Membro I: M.Sc. João Carlos Nunes Bittencourt

ASS _____
Presidente: M.Sc. Tiago Palma Pagano

Orientador: Dr. Tassio Ferreira Vale
Graduando: Tairone Conceição Dias

CRUZ DAS ALMAS, 05 DE JULHO DE 2019

Dedico este trabalho aos meus pais
pelo incentivo, apoio e motivação
desde o início de minha vida.

Agradecimentos

Agradeço a todos que contribuíram no
decorrer da minha graduação, em especial:

A Deus, que devo minha vida, saúde
e inteligência;

Aos meus pais, que sempre me apoiaram e
deram força para lutar e vencer na vida
com honestidade;

A Brenda Beatriz pelo amor, incentivo e
compreensão nos momentos difíceis;

Aos meus amigos que contribuíram direta ou
indiretamente neste trabalho, em especial,
Douglas Jamson e Joshua Passos;

Ao Prof. Dr. Tassio Ferreira Vale,
pela orientação, competência, disponibilidade
e transmissão de conhecimentos.

"Everybody in this country should learn how to program a computer...
because it teaches you how to think!"
(Steve Jobs)

Resumo

Em um contexto de desenvolvimento de software, uma das abordagens para implementar produtos de software similares com qualidade e eficiência é a Linha de Produção de Software (LPS), onde há uma identificação e tratamento sistemático da variabilidade presente nas funcionalidades dos projetos. Uma das linguagens de programação utilizadas para desenvolvimento de sistemas é JavaScript (JS). Sendo JS uma das linguagens de programação crescentes no mundo, organizações desenvolvem *frameworks* de modo a aperfeiçoá-la, e por consequência, introduzi-la nos mais diversos cenários e plataformas existentes, como é o caso do *Angular* e *Ionic*. Contudo, não foram encontradas evidências na literatura que abordem sobre formas de implementar LPS, através de técnicas de implementação de variabilidade, utilizando esta linguagem e seus *frameworks*. Este trabalho visa investigar a aplicabilidade das técnicas de implementação de variabilidade parâmetros, padrões de projeto, *frameworks* e componentes e serviços em *frameworks* JS modernos. Para realizar essa investigação, com um estudo da literatura, definiu-se critérios de avaliação para as técnicas, sendo incluído o tempo de desenvolvimento, a fim de mensurar o esforço gasto pelo desenvolvedor para a implementação e desenvolveu-se um projeto piloto utilizando essas técnicas, identificando assim o seu comportamento em LPS aplicados aos *frameworks Angular* e *Ionic*. Possuindo o conhecimento prático de implementação, aplicou-se as técnicas em um estudo de caso, onde foi possível analisar e identificar os benefícios e deficiências de cada técnica de implementação de variabilidade em um projeto de contexto real. Todas as técnicas foram aplicáveis as tecnologias *Angular* e *Ionic*, com exceção da técnica *framework* que apresentou uma limitação para compor automaticamente as *features* no processo de derivação do sistema final. Para o estudo de caso, componentes e serviços combinada como padrões de projeto foi a técnica candidata para o desenvolvimento por atender uma quantidade maior de critérios de avaliação.

Palavras-chave: *JavaScript*, Linhas de Produção de Software Orientado a *Feature*, Gerenciamento de Variabilidade.

Abstract

In a software developement context, Software Product Line (SPL) is an approach to support the implementation of similar software products with quality and efficiency, based on a systematic identification and management of the variability present in the products' features. One of the programming languages used for software development is JavaScript (JS), which is one of the fastest growing programming languages. In its ecosystem, organizations develop frameworks such as Angular and Ionic. However, there is not evidence available in the literature addressing SPL development through variability implementation techniques using JS and its related frameworks. This work aims to investigate the applicability of variability implementation techniques (e.g. parameters, design patterns, frameworks, components and services) in modern JS frameworks. In order to carry out this research, a set of evaluation criteria were defined aiming to measure the effort expended by the developer for the implementation. In addition, a pilot project was developed using the variability implementation techniques to understand their characteristics when applied to SPLs using JS with the Angular framework. With the prior implementation knowledge, the techniques were investigated in a case of study, where it was possible to analyze and identify the benefits and deficiencies of each variability implementation technique in a real context project. All techniques were applicable to the Angular and Ionic technologies, except for the framework technique that presented a limitation to automatically compose the features in the process of derivation of the final system. For the case study, components and services combined as design patterns was the candidate technique for development because it meets a larger number of quality criteria.

Keywords: JavaScript, Feature-Oriented Software Product Lines, Variability Management.

Lista de ilustrações

Lista de tabelas

Lista de siglas

LPS Linha de Produção de Software

SPL Software Product Line

JS JavaScript

HTML HyperText Markup Language

CSS Cascading Style Sheets

API Application Programming Interface

PWA Progressive Web App

IDE Integrated Development Environment

CLI Command Line Interface

Sumário

1

Introdução

1.1 Contexto

Um software padronizado é aquele desenvolvido para obedecer uma especificação única que atende às necessidades de todos os seus usuários. O uso de todas as funcionalidades por um único usuário é improvável, por isso, a falta de customização para necessidades específicas pode ocasionar na redução da satisfação pelo produto (GUTIERREZ; ALEXANDRE, 2004). Para atender casos específicos de clientes, as empresas de software oferecem o desenvolvimento de sistemas customizados, todavia, podem custar mais que o padronizado por demandar mais recursos de desenvolvimento (POHL; BOCKLE; LINDEN, 2005).

Para desenvolvimento de software customizado, uma das técnicas de engenharia de tais produtos é chamada de Linha de Produção de Software (LPS). Entende-se como linha de produção um conjunto de produtos presentes em um portfólio de uma empresa que compartilham semelhanças substanciais e que são criados a partir de peças reutilizáveis (APEL *et al.*, 2013). No caso de uma LPS, têm-se os artefatos e funcionalidades desenvolvidas previamente que podem ser reutilizadas, de forma sistemática, a cada novo sistema implementado (ALVIN, 2016).

A LPS melhora o processo de produção de novos sistemas em organizações que possuem um desenvolvimento de software customizados onde há uma similaridade entre eles. Os benefícios desta abordagem incluem aumento na qualidade das funcionalidades similares, redução no tempo desenvolvimento, no custo do sistema e na manutenção das funcionalidades (POHL; BOCKLE; LINDEN, 2005).

Dessa forma, LPS produz uma divisão entre engenharia de domínio (processo de analisar o domínio de uma linha de produtos e desenvolver funcionalidades reutilizáveis) e engenharia de aplicação (processo de desenvolver um sistema que atenda às necessidades do cliente), e entre espaço de problema (coleta de informações e problemas a serem resolvidos pela perspectiva do cliente) e espaço de solução (decisões técnicas a serem

tomadas pela perspectiva da equipe de desenvolvimento). Com a utilização de engenharia de domínio e espaço da solução, gera-se o grupo de implementação de domínio, onde têm-se a parametrização, padrões de projeto, *frameworks*, componentes e serviços, as quais são técnicas de implementação em LPS capazes de serem aplicadas nas principais linguagens de programação (APEL *et al.*, 2013).

Com base nos dados da Stack Overflow (2019), uma consulta realizada mostra que 69,7% dos desenvolvedores profissionais utilizam a linguagem de programação JavaScript JS, principalmente em sistemas web modernos onde é interpretada por navegadores de diversas plataformas como computadores, consoles de jogos, *smartphone* e *tablet*; tornando-se onipresente na história da computação (FLANAGAN, 2006). Empresas como Facebook[1], Google[2] e Ionic[3], desenvolveram o *React Native*, *Angular* e *Ionic*, respectivamente, os quais são *frameworks* JS utilizados no desenvolvimento de sistemas para navegadores web e dispositivos móveis, com o objetivo de aperfeiçoar a linguagem.

A variabilidade é um conceito essencial em LPS, pois trata-se de uma característica variável modelada para permitir o desenvolvimento de software personalizados, reutilizando artefatos predefinidos e ajustáveis (POHL; BOCKLE; LINDEN, 2005). Na literatura não foram encontradas evidências que reportem o uso de técnicas de implementação de variabilidade em *frameworks* web modernos. Contudo, torna-se relevante investigar a aplicabilidade das técnicas de implementação de variabilidade em *frameworks* JS, com foco específico nas tecnologias *Angular* e *Ionic* em um estudo de caso.

1.2 Objetivos

1.2.1 Objetivo Geral

Investigar a aplicabilidade das técnicas de implementação de variabilidade em *frameworks* web modernos baseadas na linguagem de programação JS.

1.2.2 Objetivos Específicos

- ❑ Identificar as técnicas de implementação de variabilidade em LPS aplicados em *frameworks* web modernos;
- ❑ Aplicar as técnicas de implementação de variabilidade identificadas em um estudo de caso;
- ❑ Definir os critérios para avaliação das técnicas de implementação de variabilidade;

[1] https://facebook.github.io/react-native/
[2] https://angular.io/
[3] https://ionicframework.com/

- ❏ Analisar os resultados obtidos e reportar as vantagens e desvantagens das técnicas de implementação de variabilidade.

1.3 Justificativa

Para a empresa que possui o desenvolvimento de software exclusivo para uma determinada área de negócio, oferecer um produto customizado derivado de funcionalidades reutilizáveis de outros sistemas, com preço razoável, em comparação a um sistema exclusivo, torna-se atrativo tanto para o cliente quanto para a empresa.

O planejamento para a implementação de uma LPS pode acarretar na redução de custos para desenvolver, entregar e manter o sistema, tendo em vista que a funcionalidade é igual, ou similar, para os sistemas já desenvolvidos pela empresa. Além disso, pode aprimorar a qualidade das funcionalidades, pois as mesmas são revisadas e testadas em diversos sistemas, aumentando assim as chances de detecção de falhas e correção. E, maximizar os lucros da empresa, pois com a mesma equipe de desenvolvimento mais produtos similares poderão ser desenvolvidos e entregues (POHL; BOCKLE; LINDEN, 2005).

A literatura em LPS propõe diversas técnicas de implementação de variabilidade. Entre elas, parâmetros, padrões de projeto, *frameworks*, componentes e serviços, controle de versão, *build systems*, pré-processadores, programação orientada a *feature*, programação orientada a aspecto e separação virtual de *concerns*. E, dependendo de como a variabilidade é implementada os desenvolvedores podem produzir artefatos diferentes, desde parâmetros de tempo de execução e diretivas de pré-processador à *plug-ins* e componentes (APEL *et al.*, 2013).

Não foram encontradas evidências na literatura sobre a implementação de variabilidade em LPS utilizando *frameworks* web modernos, como *Angular* e *Ionic*. Tendo em vista que esses *frameworks*, baseados na linguagem JS, estão em crescente utilização nos projetos de software em diversas plataformas, torna-se pertinente a investigação das técnicas de implementação de variabilidade visando melhorar a produtividade dos desenvolvedores.

1.4 Metodologia

A metodologia de desenvolvimento deste trabalho será dividida em:

- ❏ Revisão bibliográfica sobre as técnicas utilizadas em LPS;
- ❏ Estudo sobre a linguagem JS e os *frameworks Angular* e *Ionic*;
- ❏ Estudo e identificação das técnicas de implementação de variabilidade em LPS;

- ❏ Avaliação das técnicas de implementação de variabilidade no estudo de caso;
- ❏ Discussão sobre os resultados.

2

Revisão Bibliográfica

2.1 Linha de Produção de Software

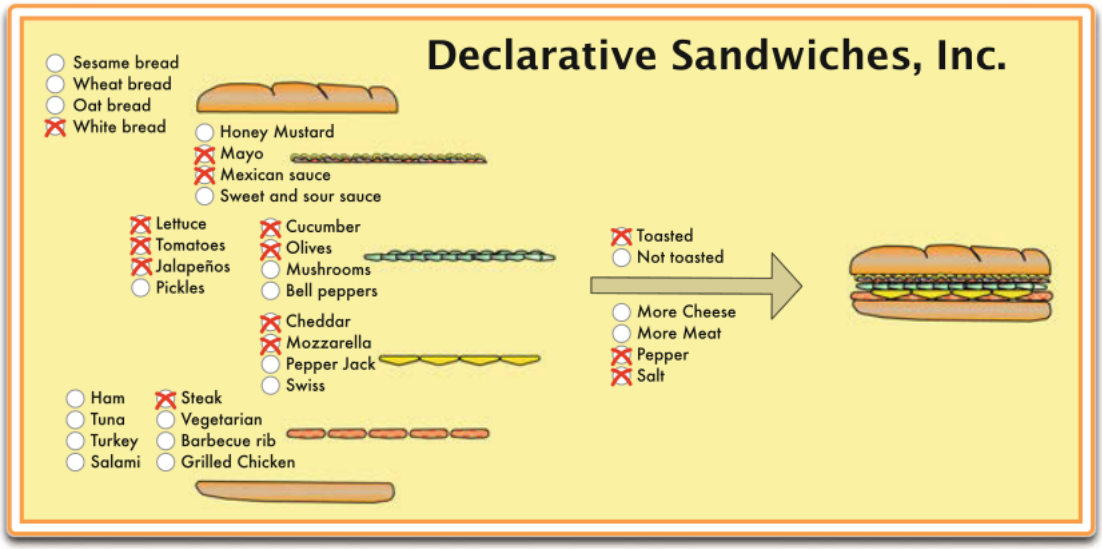
Reconhecendo que diferentes clientes têm necessidades e desejos distintos, os fabricantes começaram no século XX a aumentar a diversidade de seus produtos. Com o surgimento da linha de produtos foi possível criar a partir de um conjunto de peças reutilizáveis e semelhanças substanciais, um novo produto. Dessa forma os clientes ajustam o seu produto escolhendo entre um conjunto predefinido de opções de configuração e, os produtos são construídos em grande escala, de forma automatizada com base em peças padronizadas e reutilizáveis (APEL *et al.*, 2013).

Na Figura 1 Apel *et al.* (2013) trazem um exemplo de aplicação bem-sucedida de uma abordagem de linha de produtos na indústria de *fastfood*. Percebe-se que o cliente pode personalizar o seu produto final, sanduíche, através de um configurador que apresenta uma variedade de escolhas para pães, recheios, bem como molhos e temperos, possíveis. Conclui-se que, por apresentar peças reutilizáveis e o processo de fabricação ser o mesmo, tem-se uma variedade no produto final.

Segundo Apel *et al.* (2013) o desenvolvimento de software teve uma história bem parecida com a produção de bens físicos. Os primeiros produtos a serem desenvolvidos foram sob encomenda para hardware específicos, porém tornou-se complexo o desenvolvimento a medida do crescimento tanto pela demanda do produto quanto pela diversificação em cenários onde seria utilizado. Dessa forma, a indústria de desenvolvimento de software optou pela padronização de seus produtos, possibilitando atender a demanda dos clientes e evitando um produto caro e passível de erros, caso fossem desenvolvidos sob encomenda.

De acordo com Pohl, Bockle e Linden (2005), produtos de software feitos sob encomenda são relativamente caros. Então, na tentativa de fornecer um software que atenda a necessidade da maioria dos clientes, muitas funcionalidades são inseridas no produto, podendo torná-lo complexo, lento e com falhas. A padronização permitiu que a indústria de software fornecesse um produto acessível a um mercado amplo, porém não

Figura 1 – Configuração de um sanduíche



Fonte: Apel *et al.* (2013, p. 6)

atende a mercados menores e nem a clientes com necessidades e desejos individuais (APEL *et al.*, 2013).

Em concordância com Apel *et al.* (2013), em vez de desenvolver os sistemas do zero, eles devem ser construídos a partir de peças reutilizáveis, onde os clientes escolhem as funcionalidades e em vez de desenvolver o software sempre da mesma maneira, adaptando-o aos requisitos do cliente. Tem-se então a abordagem da LPS, cujo objetivo é fornecer produtos personalizados a custos razoáveis (POHL; BOCKLE; LINDEN, 2005). Ou seja, LPS fornece uma forma de customização em massa, desenvolvendo soluções que atendem interesses individuais com base em um portfólio de componentes de software reutilizáveis. Porém, para desenvolver o software utilizando LPS é necessário uma reutilização sistemática das funcionalidades, a partir da captura de semelhanças entre um conjunto de software e a gestão de variabilidade entre os produtos (SILVA *et al.*, 2011; BOSCH *et al.*, 2001).

2.1.1 Vantagens da utilização de LPS

A LPS consegue reunir as vantagens presentes tanto em um software desenvolvido sob encomenda quanto em um software padronizado (APEL *et al.*, 2013). As principais são destacadas na listagem a seguir:

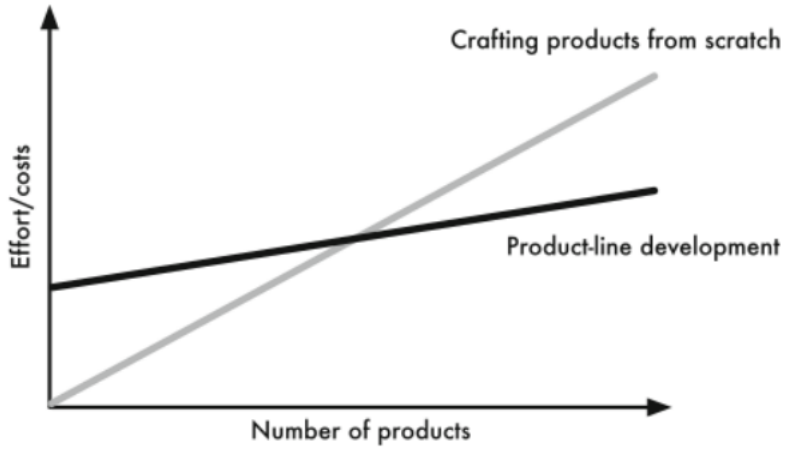
- Customização: A empresa pode fornecer para o cliente um software customizado, em vez de fornecer um software padronizado, garantindo que as reais necessidades dos clientes sejam atendidas, com características de feito sob encomenda.
- Melhor qualidade: O software sendo desenvolvido por uma LPS pode apresentar uma

qualidade melhor do que o software padronizado e o feito sob encomenda. Deve-se ao fato das funcionalidades serem verificadas sistematicamente e testadas em muitos produtos. Nem todas as combinações de funcionalidades presentes no desenvolvimento do software podem ser usadas pelos clientes, porém as funcionalidades mais utilizadas levam o produto a ser mais estável, enxuto e confiável.

- Tempo de desenvolvimento: Caso o cliente solicite algumas funcionalidades que não foram ainda criadas, desenvolver um novo software em cima de funcionalidades reutilizáveis, bem projetadas e existentes, é mais rápido do que desenvolvê-lo por completo. No entanto, é necessário uma plataforma bem projetada que atenda a expansão de novos sistemas reagindo rapidamente às mudanças do mercado.

- Custos reduzidos: O motivo essencial para introduzir LPS é a redução de custos (POHL; BOCKLE; LINDEN, 2005). Uma empresa de software consegue diminuir os custos com a equipe de desenvolvimento, pois reutilizam funcionalidades já desenvolvidas previamente, podendo então, a redução dos custo ser refletida para o cliente. Na Figura 2 apresentada por Apel *et al.* (2013), percebe-se que preparar funcionalidades reutilizáveis requer um investimento inicial significativo utilizando a LPS. No entanto, a longo prazo, quando a empresa possui uma quantidade considerável de software desenvolvidos o custo para a empresa é menor.

Figura 2 – Esforço/Custo no desenvolvimento de produtos sob encomenda x LPS



Fonte: Apel *et al.* (2013, p. 9)

2.1.2 Desenvolvimento de uma LPS

As principais preocupações para um processo de desenvolvimento utilizando LPS são as *features*. Conceito de *feature* corresponde a um aspecto, qualidade ou característica visível ao usuário final de um software (KANG *et al.*, 1990). Então, em uma LPS, *features* são utilizadas para especificar e comunicar semelhanças e diferenças dos produtos entre as

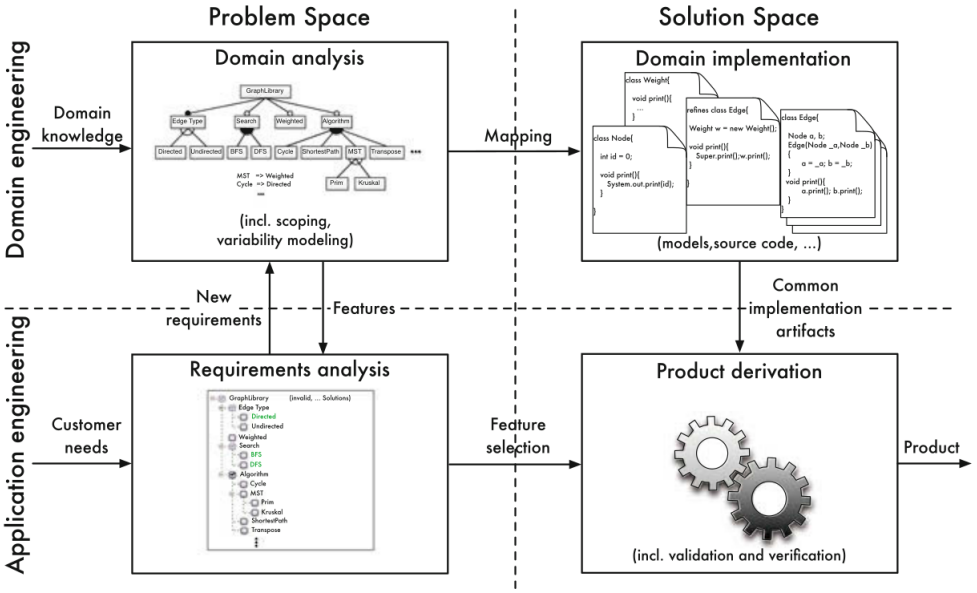
partes interessadas e para orientar a estrutura, reutilização e variação em todas as fases do ciclo de vida do software (APEL *et al.*, 2013).

Dessa forma, o configurador ou portfólio do produto de uma linha de produtos é definido por *features* e suas relações, facilitando a escolha do cliente. Deste modo, o produto final, neste caso o software, é um conjunto de seleções de *features* válidas (APEL *et al.*, 2013).

Atualmente o desenvolvimento de software utiliza um processo tradicional. Inicialmente ocorre a coleta dos requisitos, a criação do projeto e a implementação que podem acontecer em fases separadas, consecutivas ou em ciclos ágeis. Para LPS é necessário que a empresa pense diferente sobre o desenvolvimento de software, analisando a variabilidade dos sistemas que são semelhantes, mas não idênticos (APEL *et al.*, 2013).

Um processo de desenvolvimento de LPS depende diretamente do tratamento explícito da variabilidade e da reutilização sistemática de funcionalidades para implementação. A Figura 3 apresenta as características específicas da LPS onde se tem a separação entre a engenharia de domínio e a engenharia de aplicação, e entre o espaço do problema e o espaço da solução (APEL *et al.*, 2013).

Figura 3 – Processo de engenharia para LPS



Fonte: Apel *et al.* (2013, p. 20)

A engenharia de domínio é o processo de analisar o domínio de uma linha de produtos e desenvolver funcionalidades reutilizáveis. Dessa forma, ela prepara a estrutura para ser utilizada em vários produtos da LPS. Em contrapartida, a engenharia de aplicação tem o objetivo de desenvolver um software específico para as necessidades de um determinado cliente. Corresponde então, ao processo de desenvolvimento de um software único que

reutiliza funcionalidades da engenharia de domínio sempre que possível. Tem como alvo o desenvolvimento com reutilização. Portanto, enquanto a engenharia de domínio visa o desenvolvimento para reutilização e é executada apenas uma vez, a engenharia de aplicação foca no desenvolvimento com reutilização e é executada para cada novo software (APEL *et al.*, 2013).

O espaço do problema toma a perspectiva do cliente e seus problemas a serem resolvidos, como os requisitos. Já o espaço da solução representa as perspectivas da equipe de desenvolvimento, caracterizando-o com termos técnicos, que inclui nomes de funções, classes e parâmetros do programa. O espaço da solução também cobre o projeto, a implementação, validação e verificação de *features* e suas combinações de maneiras adequadas para facilitar a reutilização sistemática (APEL *et al.*, 2013).

Com isso pode-se extrair quatro grupos de tarefas no desenvolvimento da LPS: a análise de domínio, onde decide quais produtos devem ser cobertos pela LPS e quais *features* são relevantes e devem ser implementados como funcionalidades reutilizáveis. A análise de requisitos que investiga as necessidades do cliente como parte da engenharia de aplicação, então os requisitos de um cliente são mapeados para uma seleção de *features*, com base nas *features* identificadas durante a análise de domínio. A implementação de domínio que é o processo de desenvolvimento de funcionalidades reutilizáveis que atendem as *features* identificadas na análise de domínio. E, a derivação de produto que é a etapa de produção da engenharia de aplicação, onde as funcionalidades reutilizáveis são combinados de acordo com os resultados da análise de requisitos (APEL *et al.*, 2013).

2.1.3 Modelagem de *feature*

A modelagem de variabilidade é um passo crucial no desenvolvimento da LPS. Segundo Apel *et al.* (2013), existem muitas abordagens diferentes de modelagem de variabilidade, porém a mais popular é a modelagem de *features*.

A modelagem de *features* apesar de acontecer na análise de domínio, os seus resultados desempenham um papel central em outras fases do desenvolvimento da LPS. Então, um modelo de *features* especifica o conjunto de produto válidos, ou seja, documenta quais os relacionamento entre as *features* e define quais seleções de *features* são válidas.

2.1.4 Taxonomia para classificação de variabilidade

Sabe-se que variabilidade é a capacidade de derivar produtos diferentes a partir de um conjunto de *features* a serem escolhidos pelo cliente. Com isso, Apel *et al.* (2013) descrevem três critérios de classificação, que são a *binding time*, tecnologia e representação.

Binding time é um critério de classificação de variabilidade que decide quais *features* devem ser incluídas no software. Esse processo é decidido antes ou em tempo de compilação,

no tempo de carregamento e no tempo de execução, sendo que nesta última as decisões podem ser tomadas e alteradas durante a execução do programa (APEL *et al.*, 2013).

Cada *binding time* possui suas vantagens e desvantagens. O *binding time* de compilação é mais otimizado, pois todo o código desnecessário pode ser removido do produto, reduzindo a sobrecarga no tempo de execução e consumo de memória. No entanto, uma vez que o software é criado e instalado, ele não é mais variável (CHENG *et al.*, 2009 apud APEL *et al.*, 2013).

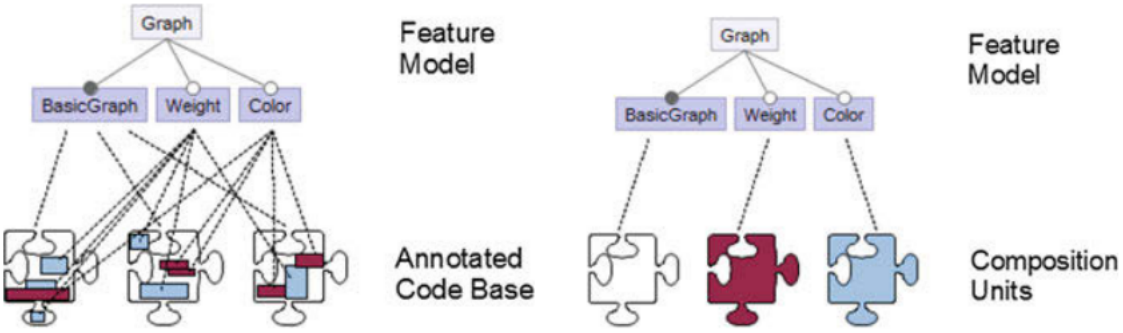
Já um software que possui o *binding time* de carregamento é mais flexível para reconfiguração, pois em vez de recompilar um novo produto após cada alteração, os usuários podem modificar parâmetros e reiniciar o mesmo software. No caso do *binding time* de execução, nem é necessário reiniciar o software podendo mudar durante a execução do programa. No entanto, ambos os mecanismos apresentam uma sobrecarga de memória e desempenho, pois todas as variações são compiladas em um único código binário de forma que não ocorram inconsistências em tempo de execução (APEL *et al.*, 2013).

O critério de tecnologia possui as abordagens *language-based* e *tool-based*. A abordagem *language-based* utiliza mecanismo fornecidos pela linguagem de programação, assim a implementação e gerenciamento de *features* e a variabilidade estão localizadas no código fonte do software. Já a abordagem *tool-based* usa uma ou mais ferramentas externas para implementar *features* no código e para controlar o processo de derivação do produto (APEL *et al.*, 2013).

Por fim, o terceiro critério é a representação por anotação e por composição. Em abordagens baseadas em anotação, todo o código das *features* são inseridas e marcadas, como mostra a Figura 4a. Durante o processo de derivação do software, os códigos que pertencem a uma *feature*, ou combinação delas, não marcadas, é removida, em tempo de compilação, ou ignoradas, em tempo de execução. Por conta disso, essa abordagem de anotação suporta a variabilidade negativa, onde o código é removido sob demanda. Há um crescimento na adoção dessa abordagem por possuir uma facilidade de utilização e pelos ambientes de programação oferecerem suporte nativo (APEL *et al.*, 2013).

As abordagens baseadas em composição, como *frameworks* e componentes localizam o código pertencente a uma *feature*, ou combinação delas, contêiner ou módulo, e implementam na forma de unidades compostas, idealmente uma unidade por *feature*. No processo de derivação do produto, todas as unidades de todas as *features* selecionadas e combinações válidas são compostas para criar o software, podendo ser visualizadas na Figura 4b. Por conta disso, essa abordagem de composição suporta a variabilidade positiva, que a possibilidade do código ser adicionado sob demanda (APEL *et al.*, 2013). Conclui-se que estabelecendo um foco adequado em um domínio específico com um escopo e técnica de implementação de variabilidade bem definido, obtem-se o sucesso para a introdução de LPS em uma empresa de software.

Figura 4 – Abordagem baseada em anotação e baseada em composição para LPS



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

Fonte: Apel *et al.* (2013, p. 52)

A Tabela 1 apresenta uma classificação de técnicas de implementação de variabilidade em LPS quanto aos critérios de classificação. Nota-se que todas as abordagens de implementação possuem todos os critérios de classificação, com isso, conclui-se que cada técnica é composta por mais de um critério.

Tabela 1 – Classificação de abordagens de implementação em LPS

	Binding time		Tecnologia		Representação	
	Compilação	Carregamento (e execução)	Language-based	Tool-based	Anotação	Composição
Parâmetros		x	x		x	
Padrões de Projeto	(x)[1]	x	x			x
Frameworks	(x)[1]	x	x			x
Componentes	x	x	x	x		x
Controle de Versão	x			x		x
Build Systems	x			x	(x)[3]	x
Pré-processadores	x			x	x	
Programação Orientada a *Feature*	x	(x)[2]	x			x
Programação Orientada a Aspecto	x	(x)[2]	x			
Separação Virtual de *Concerns*	x			x	x	

Fonte: (APEL *et al.*, 2013, 61)

[1] Tanto padrões de projeto quanto *frameworks* suportam a variabilidade do tempo de compilação e do tempo de execução, mas, no caso da variabilidade do tempo de compilação, eles induzem ainda uma sobrecarga de tempo de execução. [2] Tanto a programação orientada a *feature* e programação orientada a aspecto suportam *binding time* de carregamento ou execução, mas as ferramentas correspondentes são bastante experimentais. [3] Um caso específico de atribuição de um arquivo inteiro para uma determinada *feature*.

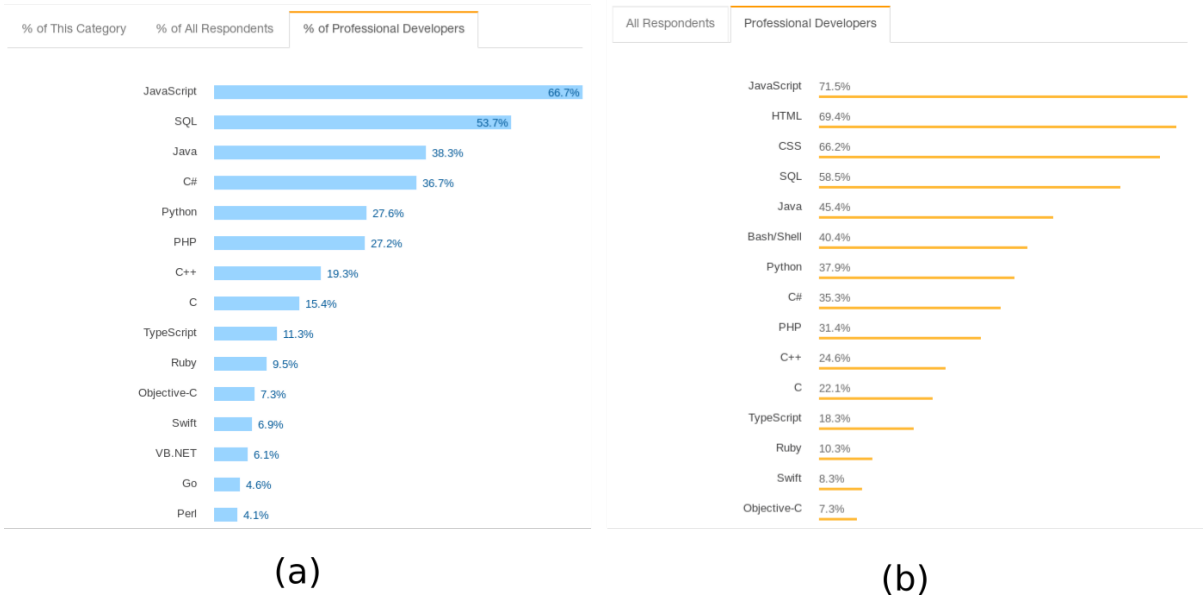
2.2 *Frameworks JavaScript* Modernos

Uma parte considerável dos sites e sistemas modernos presentes na web, usam JS. Segundo Flanagan (2006), JS é uma linguagem de programação da web de alto nível, dinâmica, interpretada e não tipada, ideal para programação orientada a objetos e funcional. Pelo fato de ser uma das primeiras linguagens da web a surgir, todos os navegadores modernos incluem interpretadores JS.

Inicialmente, JS era apenas uma linguagem de *script*, que com o tempo tornou-se uma linguagem de uso geral, robusta e eficiente. Mas, antes de ter o nome *JavaScript*, a linguagem apresentou outras denominações como: *"ECMAScript"*, depois *"JScript"*, e por questões de marketing, já que no mesmo período a linguagem *Java* tornava-se popular no desenvolvimento de sistemas para desktop, adotou-se o nome de *JavaScript*. Um outro motivo é somente a sintaxe do JS ser derivada da linguagem *Java* (FLANAGAN, 2006).

Em 2017, uma pesquisa, realizada pelo site *Stack Overflow*, consultou 27.612 desenvolvedores que responderam perguntas sobre as linguagens e tecnologias que aprenderam, que construíram suas carreiras profissionais ou que demandam no mercado de trabalho. Cerca de 66.7% dos usuários responderam que utilizaram JS para desenvolvimento profissional, como pode ser visualizada na Figura 5a. Um ano depois, uma nova consulta foi realizada com 73.248 desenvolvedores e esse percentual aumentou para 71.5%, como mostra a Figura 5b.

Figura 5 – Tecnologias mais populares de 2017 e 2018



Fonte: Stack Overflow (2017), Stack Overflow (2018)

A pesquisa da Stack Overflow (2019) consultou 72.525 desenvolvedores e, JS continua em primeiro lugar, com 69.7% das respostas. Percebe-se que por três anos

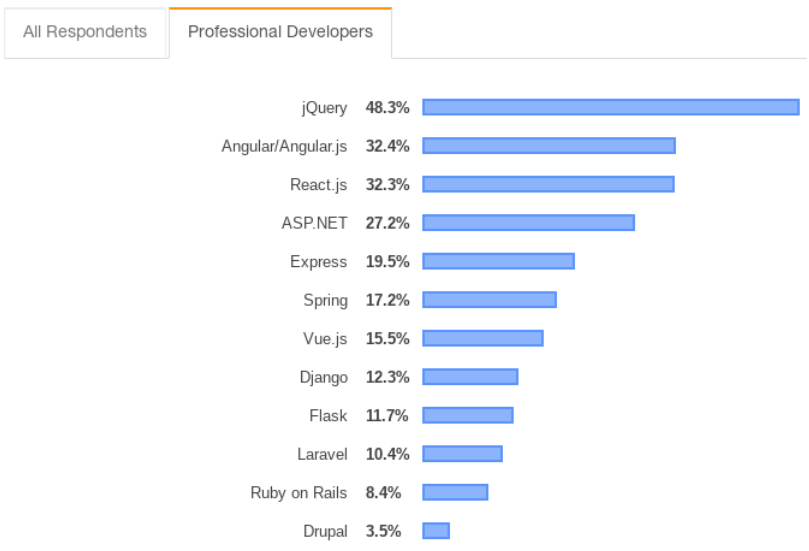
consecutivos JS é a linguagem mais utilizada no mundo e em concordância com Flanagan (2006), JS tornou-se a linguagem de programação onipresente da história da computação. Diante dessa crescente, utilizadores optam por desenvolver seus software em *framework* JS.

Frameworks são "estruturas" que possuem uma *Application Programming Interface (API)* de abstração mais alta, consequentemente uma nova sintaxe de programação, sobre uma API padrão. As principais vantagens são o desenvolvimento de um sistema com poucas linhas de código, pois os *frameworks* possuem um nível de linguagem mais alto, e o tratamento de problemas que a linguagem da API padrão possui, como compatibilidade, segurança e acessibilidade (FLANAGAN, 2006).

O *AngularJS*, ou simplesmente *Angular*, é um *framework* JS criado pelo Google de fácil manutenção para o desenvolvimento de sistemas web. Sua filosofia envolve que todo código declarativo é melhor do que o código imperativo para o desenvolvimento de interfaces do usuário e conexões entre diferentes componentes de aplicativos da web (JAIN; BHANSALI; MEHTA, 2015).

O Angular abrange a extensão do HyperText Markup Language (HTML) para um formato mais expressivo e legível, permitindo uma sincronização do JS com a marcação do HTML, evitando assim a atualização manual das informações ou da visualização de componentes (JAIN; BHANSALI; MEHTA, 2015). Como apresentado na Figura 6, Angular é o segundo *framework* mais utilizado por desenvolvedores profissionais atingindo 32.4% de 55.079 consultas realizadas, ficando atrás de *jQuery*, que também é um *framework* JS utilizado no *front-end*.

Figura 6 – Bibliotecas, *frameworks* e ferramentas



Fonte: Stack Overflow (2019)

Ionic é um *framework* que promove uma estrutura de interface de usuário criada

em tecnologias web para o desenvolvimento de aplicativos móveis híbridos ou Progressive Web App (PWA) (IONIC, 2018). Entende-se por aplicativo móvel híbrido um sistema que é construído utilizando tecnologias web, como HTML, Cascading Style Sheets (CSS) e JS que permite a utilização de recursos nativos do dispositivo móvel, como câmera, gps e acelerômetro (GRIFFITH, 2017).

O *Ionic* também pode ser considerado como uma combinação de tecnologias, sendo elas, o próprio *Ionic Framework* para a interface do usuário, o *Angular* para a manipulação das interfaces e o *Apache Cordova* que permite a utilização de recursos nativos do dispositivo transformando o aplicativo da web em um híbrido (GRIFFITH, 2017).

A principal vantagem do *Ionic* é a organização ao desenvolver um único software e compilar para os principais sistemas operacionais móveis do mercado, *iOS* e *Android* (GRIFFITH, 2017; RASMUSSEN, 2018). Apesar dos pontos negativos serem no tempo de compilação para cada sistema operacional móvel, há pouca documentação e desempenho abaixo, comparado ao aplicativo nativo. O *Ionic* adequa-se perfeitamente com LPS, onde o principal foco é a reutilização sistemática de *features* (RASMUSSEN, 2018).

Typescript é uma linguagem desenvolvida pela *Microsoft*[1] utilizada para facilitar o desenvolvimento de projetos em *Angular* e *Ionic*, pois incorpora conceitos como tipagem estática e orientação a objetos que são complicados de desenvolver e difíceis de compreender em JS (FAIN; MOISEEV, 2016).

2.3 Técnicas de Implementação de Variabilidade

As técnicas de implementação de variabilidade voltadas para linguagens de programação são: parâmetros, padrões de projeto, *frameworks*, componentes e serviços (APEL *et al.*, 2013). Nas próximas subseções há uma definição e explicação da implementação dessas técnicas em LPS.

2.3.1 Parâmetros

Parâmetros utilizam uma maneira simples de implementar variabilidade através de estruturas condicionais, como *if* e *switch*, para alterar o fluxo de controle e execução do programa. Ou seja, as instruções condicionais determinam um fluxo com base nos parâmetros de configuração passados para métodos ou módulos durante a execução do programa, tendo o efeito imediato ou em alguns casos exigindo uma reinicialização do software (APEL *et al.*, 2013).

Normalmente, utilizam-se variáveis globais do tipo *booleano* por *feature* para receberem os parâmetros de configuração e para serem analisadas nas estruturas condicionais,

[1] https://www.typescriptlang.org/

no entanto, desencorajam uma solução modular. Caso a implementação seja disciplinada, obtem-se uma facilidade na rastreabilidade da *feature*, pois a relação entre *feature* e parâmetros é expressa por convenções de nomenclatura (APEL *et al.*, 2013).

2.3.2 Padrões de Projeto

Design patterns são tipicamente dados usando notações convencionais que não possuem uma semântica bem definida (MIKKONEN, 1998). Eles captam a intenção de um projeto identificando objetos, suas colaborações e a distribuição de responsabilidade (PREE; GAMMA, 1995). De acordo com Apel *et al.* (2013), os padrões de projeto conhecidos como *observer*, *template-method*, *strategy* e *decorator* são padrões adequados para a implementação de variabilidade.

2.3.2.1 Padrão *Observer*

O padrão *observer* consiste em implementar uma forma de tratamento distribuído de eventos, onde um sujeito notifica todos os *observers* registrados sobre alterações em seu estado. Assim, sua estrutura compõe em uma interface *observer*, que contém um ou mais métodos que são invocados pelo sujeito em mudanças de estado; *observers* concretos, implementando a interface *observer* e reagindo a mudanças pelo sujeito; e, um sujeito para o qual os *observers* podem se registrar, quando necessário enviando eventos para todos os *observers* registrados (APEL *et al.*, 2013).

Com o padrão *observer* há uma dissociação entre o sujeito e os *observers*, adicionando flexibilidade para adicionar ou remover *observers* posteriormente neste contexto, cada *feature* pode ser implementado como um *observer*. Assim, em LPS a variabilidade é garantida registrando ou não os *observers* (APEL *et al.*, 2013).

2.3.2.2 Padrão *Template-Method*

O padrão *template-method* define um esqueleto de classe abstrata, deixando que particularidades sejam especificadas por uma subclasse que à herdar. Assim, sua estrutura consiste em uma classe abstrata, contendo métodos, variáveis comuns ao sistema; e diferentes subclasses que através de polimorfismo fornecem implementações distintas dessas etapas substituindo um ou vários métodos, mudando assim o comportamento do programa (APEL *et al.*, 2013).

No desenvolvimento de LPS as *features* podem ser implementadas como subclasses, especialmente se forem *features* alternativas, pois o algoritmo difere apenas em detalhes menores, compartilhando o esqueleto do algoritmo de uma classe abstrata. *Features* opcionais também podem ser implementadas, mas como em algumas linguagens há uma limitação de herança, esse padrão fica inviável (APEL *et al.*, 2013).

2.3.2.3 Padrão *Strategy*

No padrão *strategy* a variabilidade é semelhante ao padrão *template-method*, pois usa delegação em vez de herança. Ou seja, consiste em uma interface que descreve a funcionalidade, através de método, que pode ser fornecida aos cliente, as classes concretas que implementam a interface e o contexto que implementa o algoritmo principal, além de conter um mecanismo de *callback* (APEL *et al.*, 2013).

Assim como o padrão *template-method*, as *features* alternativas são mais adequadas do que as *features* opcionais para o padrão *strategy*, tornando possível a seleção de uma *feature* por vez dentro um conjunto de *features*. Utilizando este padrão no desenvolvimento de LPS, as *features* correspondem a diferentes implementações de métodos (APEL *et al.*, 2013).

2.3.2.4 Padrão *Decorator*

O padrão *decorator* possui um mecanismo baseado em delegação para estender objetos de forma flexível, ou seja, objetos de uma classe podem ser estendidos com comportamento adicional em tempo de execução, assim várias extensões podem ser combinadas (APEL *et al.*, 2013).

A estrutura do padrão *decorator* consiste em uma interface do componente que descreve seu comportamento, uma implementação concreta do componente, uma classe abstrata de *decorator* e uma ou mais implementações concretas do *decorator* que são representadas pelas *features*. Na LPS esse padrão é adequado para implementar *features* opcionais e grupos de *features* dos quais várias *features* podem ser selecionadas (APEL *et al.*, 2013).

2.3.3 *Frameworks*

Um *framework* é uma estrutura de base reutilizável, colaborativa ou não, que pode ser estendida e adaptada para resolver um conjunto relacionado de problemas suportando uma granularidade maior do que as de classes e fornecendo pontos explícitos para extensões, frequentemente chamados de *plug-ins*, nos quais desenvolvedores podem expandi-lo (APEL *et al.*, 2013).

Atualmente, *frameworks* com *plug-ins* são os mais desenvolvidos na LPS, pois idealmente, cada *feature* corresponde a um *plug-in* montando o programa final conforme a seleção de features realizada, tornando um processo de composição (APEL *et al.*, 2013).

2.3.3.1 *Framework* Caixa Branca

Framework caixa branca consiste em um conjunto de classes concretas e abstratas, assemelhando-se com o padrão *template-method*, onde os desenvolvedores implementam

ou sobrescrevem em um subclasse expandindo o *framework* caixa branca. É denominada caixa branca, pelo fato da equipe de desenvolvimento precisar identificar os métodos e entender os componentes internos do *framework*.

Dessa forma, há um *trade-off* entre flexibilidade ao adicionar extensão e esforço de compreensão do sistema, pois o *framework* caixa branca permite uma mudança do comportamento existente a partir de implementações adicionais de extensões imprevistas. No entanto, os desenvolvedores necessitam de uma compreensão detalhada e não encapsulada das extensões do *framework*.

Na LPS cada *feature* é considerada como extensão, que por consequência é caracterizada como uma subclasse de uma determinada classe, limitando a mistura e combinação de várias extensões. Contudo, como o padrão *template-method*, os *framework* caixa branca são mais adequados para implementar *features* alternativas, sendo que a principal diferença é que *framework* podem ser desenvolvidos e disponibilizados por terceiros.

2.3.3.2 *Framework* Caixa Preta

Frameworks caixa preta seguem os padrões de *strategy* e *observer*, separando os códigos entre *framework* e extensões por meio de interfaces, permitindo a implementação e compilação separadamente denominada de *plug-in*. É conhecida como caixa preta, pelo fato dos desenvolvedores necessitarem conhecer apenas a interface, descartando entender a implementação interna do *framework* (APEL *et al.*, 2013).

Como os desenvolvedores podem adicionar *plug-ins* apenas aos pontos de acesso previstos no *framework*, há uma limitação da flexibilidade. Por outro lado, essa limitação permite o desacoplamento de extensões provocando um entendimento e utilização mais simples, pois pouco código da interface deve ser entendida, além de incentivar o desenvolvimento e implementação de *plug-ins* separada (APEL *et al.*, 2013).

Teoricamente, no desenvolvimento de LPS, cada *feature* é implementada por um *plug-in* e depois combinados ao *framework*, permitindo uma automação na geração do produto final. O *framework* e *plug-ins* possuem uma evolução independente, uma vez que as interfaces do *plug-in* permanecem inalteradas (APEL *et al.*, 2013).

2.3.4 Componente e Serviços

Segundo Apel *et al.* (2013), componente é uma unidade de composição com interfaces contratualmente especificadas e contendo dependências de contexto explícita. Ou seja, um componente fornece sua funcionalidade por meio de uma interface, tendo sua implementação interna encapsulada, formando uma unidade modular e reutilizável, onde um componente é considerado uma *feature*. Nessa técnica, tem-se a abordagem de composição, onde um componente pode ser composto por outros componentes em diferentes combinações.

Ao construir um programa, desenvolvedores podem implementar e implantar seus próprios componentes de forma independente e, caso necessário, compor componentes de terceiros, pois um componente é independente de uma aplicação ou linha de produto específica. Com isso, pode-se comparar componentes com *plug-ins*, apenas se eles e suas interfaces forem projetadas para os mesmos padrões. No entanto, eles se diferenciam, pois implementações baseadas em componentes não são projetadas para serem compostas automaticamente (APEL *et al.*, 2013).

Na LPS determinar quando criar um componente reutilizável é uma decisão importante de projeto realizada na análise de domínio que ajuda a decidir como dividir o código em componentes. Se um componente reutilizável é grande e fornece muita funcionalidade torna-se fácil de integrar e utilizar, porém eventualmente em alguma aplicação o componente pode não se encaixar. Em contraste, pode-se criar pequenos componentes reutilizáveis que podem ser combinados de forma flexível, porém a quantidade de conexões que deve ser feita entre os componentes e o código base torna-se desestimulante. Dessa forma, os desenvolvedores precisam obter um equilíbrio entre implantar um componente que forneça funcionalidades, mas que seja pequeno suficiente para ser reutilizado em muitos contextos (APEL *et al.*, 2013).

O desenvolvimento de componentes na LPS é adequada se a seleção de *features* for realizada por desenvolvedores e não por clientes e, se o número de produtos finais for reduzido, pois a probabilidade de algum componente reutilizável não se adequar a um contexto é eliminada (APEL *et al.*, 2013).

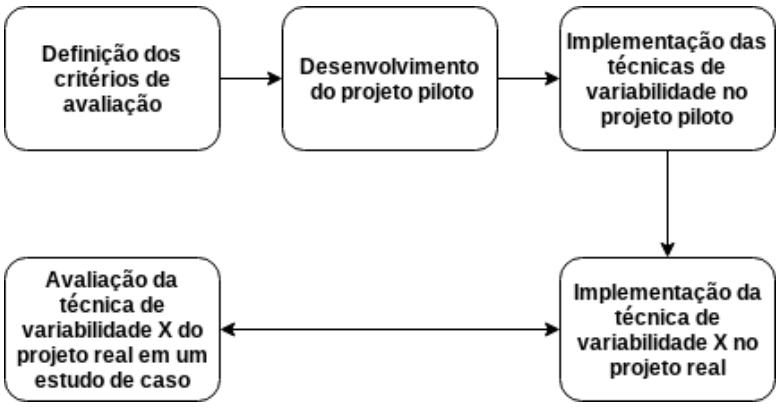
Serviço por sua vez é semelhante ao componente, pois encapsula a funcionalidade da *feature* por trás de uma interface, porém padronização, interoperabilidade e distribuição são fatores valorosos da técnica. Um outro diferencial é que serviços escritos em linguagens diferentes podem interagir entre si, pois sua comunicação é padronizada através de protocolos ou convenções (APEL *et al.*, 2013).

3

Design de Pesquisa

Esta pesquisa foi executada a partir de um projeto contendo cinco passos. São eles: (i) definição dos critérios de avaliação, (ii) desenvolvimento do projeto piloto, (iii) implementação das técnicas de variabilidade no projeto piloto, (iv) implementação da técnica de variabilidade X no projeto real e (v) avaliação da técnica de variabilidade X do projeto real em um estudo de caso, como podemos ver na Figura 7.

Figura 7 – Fluxograma de execução do design de pesquisa

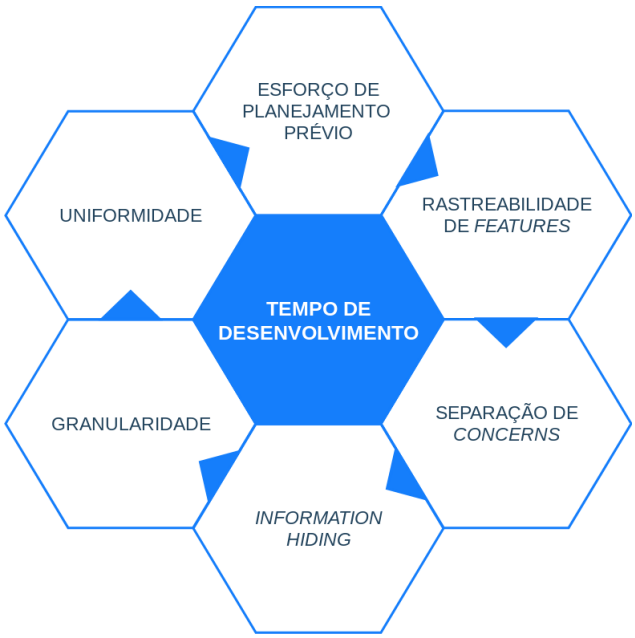


Para realizar uma análise comparativa entre as técnicas de implementação de variabilidade sobre o projeto de contexto real do estudo de caso, inicialmente definiu-se os critérios de avaliação. Sendo o foco de avaliação do trabalho a linguagem JS e de acordo com a taxonomia definida por Apel *et al.* (2013) para técnicas aplicáveis a linguagem, a tecnologia *language-based* foi utilizada para ilustrar a implementação das técnicas de variabilidade. Desenvolveu-se o projeto piloto e posteriormente implementou cada técnica de variabilidade neste projeto. Para obter maior rigor e relevância na avaliação das técnicas de implementação de variabilidade sobre os critérios de avaliação, foi projetado e executado um estudo de caso utilizando um projeto de contexto real para a implementação das técnicas, orientado por um protocolo de estudo de caso.

3.1 Critérios de avaliação

Os critérios escolhidos e descritos nas subseções seguintes e mostrada na Figura 8, são critérios de avaliação adotados em organizações e equipes de desenvolvimento, sendo uma estratégia para definir qual a técnica de implementação de variabilidade optar para o desenvolvimento de software orientado a *feature* em um determinado contexto. Com isso Apel *et al.* (2013) apresentam seis critérios de avaliação que as técnicas de implementação em LPS devem satisfazer: (i) baixo esforço no pré-planejamento, (ii) rastreabilidade de *features*, (iii) separação de *concerns*, (iv) *information hiding*, (v) granularidade e (vi) uniformidade. Como esses critérios possuem metas conflitantes, é provável não obter todos ao mesmo tempo. Ao final, este trabalho adicionou como critério de avaliação o tempo de desenvolvimento, permitindo mensurar o esforço de implementação executado pelo desenvolvedor.

Figura 8 – Critérios de avaliação



3.1.1 Esforço de pré-planejamento

Todo processo de desenvolvimento de LPS requer de um pré-planejamento. O principal objetivo do esforço de pré-planejamento é facilitar a antecipação de possíveis requisitos das *features* que podem ser variáveis e reutilizáveis, entretanto, nem todas as features e variações podem ser antecipadas (APEL *et al.*, 2013).

Uma técnica de implementação de LPS ideal, tenta minimizar o esforço de pré-planejamento. Cada técnica tem um desempenho diferente de acordo com o grau de esforço dedicado ao pré-planejamento. Ou seja, algumas técnicas de implementação promovem mudanças onde o pré-planejamento necessário é mínimo, enquanto que outras técnicas

exigem atividades substanciais de pré-planejamento. Em alguns casos, se as informações antecipadas sobre as variações são insuficientes, torna-se impossível o desenvolvimento da *feature* (APEL *et al.*, 2013).

Em geral, antecipar variações é um pré-requisito adequado em LPS, cujo objetivo é promover mudanças com pouco esforço adicionando ao projeto um importante critério de avaliação.

3.1.2 Rastreabilidade de *features*

Segundo Apel *et al.* (2013), rastreabilidade de *feature* é a capacidade de rastrear uma *feature* do espaço do problema para o espaço de solução, sendo uma das propriedades importantes das técnicas de implementação de variabilidade da LPS.

Cada técnica fornece diferentes níveis de suporte para rastreabilidade. No entanto, pode-se afirmar que toda linguagem baseada em composição, a rastreabilidade de *feautres* é simplificada, se na sua implementação há uma unidade de composição por *feature* (APEL *et al.*, 2013).

3.1.3 Separação de *concerns*

Separação de *concerns* visa uma modularização do sistema, decompondo-o em partes semanticamente coesas (DIJKSTRA, 1982; APEL *et al.*, 2013). Ou seja, separar as *features* entre o projeto e o código tornando explícita a relação entre as *features* e os artefatos de implementação, facilitando para os desenvolvedores as tarefas de manutenção e evolução do software (APEL *et al.*, 2013).

Para separar *concerns* alguns mecanismos foram desenvolvidos obedecendo estruturas hierárquicas ou estrutura de blocos, como procedimentos, módulos e classes. No entanto, existem *concerns* que possuem uma relação estrutural entre dois outros *concerns*, conhecidos como *concerns* transversais (APEL *et al.*, 2013).

Com o uso de estruturas hierárquicas, os desenvolvedores decompõem o software apenas de uma maneira, evitando idealmente código disperso e emaranhado. Na prática, podem conter *concerns*, e até *concerns* transversais, que não se alinham com a decomposição adotada provocando dispersão e emaranhamento no código. No entanto, não significa que o projeto ou a decomposição foi ruim, mas simplesmente inevitável (APEL *et al.*, 2013).

Ter a capacidade de separar *concerns* em implementações coesas é um importante critério de avaliação, sabendo que as *features* são os *concerns*, em alguns casos *concerns* transversais, de interesse primário na LPS (APEL *et al.*, 2013).

3.1.4 *Information hiding*

O *information hiding* é a separação de um módulo em uma parte interna e externa, sendo que a parte interna é a implementação da *feature* e permanece oculta, enquanto que a parte externa é a interface do módulo que especifica como o módulo interage com o resto do sistema. Assim, permite com que os desenvolvedores obtenham um raciocínio modular do sistema, podendo ou não conhecer os componentes internos dos módulos (APEL *et al.*, 2013).

Idealmente, os desenvolvedores são orientados a projetar interfaces pequenas e claras. Quando há uma separação das *features*, os aspectos internos da implementação são ocultados, sendo explícito apenas as interfaces de comunicação. Essa abordagem, evita que os desenvolvedores conheçam toda a implementação de todas as *features*; possibilitando que as equipes possam trabalhar com diferentes *features* e interligá-las com base em interfaces acordadas (APEL *et al.*, 2013).

Negligenciando o *information hiding* e contando apenas com a separação de *concerns* em artefatos coesos, ainda obtem-se uma modularização e um código sem dispersão ou emaranhado, mas não há garantias. O *information hiding* permite um facilitador na navegação e raciocínio do código e, no pior dos casos os desenvolvedores conheceriam todo o sistema (APEL *et al.*, 2013).

Portanto, o *information hiding* é um critério de avaliação pelo qual se deve levar em consideração nas técnicas de implementação de variabilidade em LPS.

3.1.5 Granularidade

Segundo Apel *et al.* (2013), granularidade refere-se à estrutura hierárquica de um artefato sob uma técnica de implementação de variabilidade específica. Dessa forma, quando as alterações acontecem no topo da estrutura hierárquica, a técnica tem como característica granularidade grossa. Em contraste, mudanças em níveis mais baixos apresentam granularidade fina, ou até mesmo granularidade média. Quando uma *feature* envolve a inclusão de um novo arquivo ou classe a um determinado programa, apresenta característica de granularidade grossa. Quando envolve a inclusão de um novo membro ou método a uma determinada classe, pode-se considerar uma granularidade média. E, quando há uma inclusão de instruções a um determinado bloco ou método é uma granularidade fina ou baixa.

Normalmente, abordagens baseadas em anotações suportam mudanças e interações de granularidade fina do que abordagens baseadas em composição. Isso porque na composição as *features* são implementadas ao nível de classes e interfaces, já em anotação o código de diferentes *features* é misturado na base de código comum (APEL *et al.*, 2013).

3.1.6 Uniformidade

Apel *et al.* (2013) consideram artefatos como sendo tanto a implementação das *features* no código quanto a criação de documentos de requisitos, descrições de arquitetura, modelos de projeto e desempenho, que são chamados de artefatos *noncode*. Com isso, define-se uniformidade como sendo a capacidade de implementar artefatos para qualquer linguagem de programação e até mesmo artefatos *noncode*, sendo um importante critério de avaliação para as técnicas de implementação de variabilidade da LPS. Portanto, uniformidade é conseguir utilizar uma técnica de implementação que seja aplicável aos diferentes artefatos existentes para o desenvolvimento das *features* (APEL *et al.*, 2013).

A Tabela 2 apresenta quais critérios de avaliação são satisfeitos pelas técnicas de implementação de variabilidade com base na literatura. Percebe-se que cada critério de avaliação apresenta objetivos conflitantes, concluindo ser impossível obter todos os critérios ao mesmo tempo com as técnicas apresentadas.

Tabela 2 – Características das técnicas de implementação de variabilidade quanto aos critérios de avaliação em LPS

	Parâmetros	**Padrões de Projeto**	***Frameworks***	**Componentes e Serviços**
Pré-planejamento	baixo	alto	alto	alto
Rastreabilidade	difícil	possível	possível	possível
Separação de *concerns*	difícil	nativo	nativo[1]	nativo[1]
Information hiding	irrealizável	nativo	nativo[2]	praticável
Granularidade	fina	grossa	grossa	grossa
Uniformidade	parcial	parcial	parcial	total

[1] Apenas em aplicações em que não há combinação entre as *features* alternativas e *features* opcionais. [2] Nativo para *frameworks* caixa preta.

3.1.7 Tempo de desenvolvimento

Apesar do tempo de desenvolvimento não ser um critério de avaliação especificado por Apel *et al.* (2013), mensurou-se aproximadamente o esforço de desenvolvimento, em unidade de tempo, que cada técnica de variabilidade levou para ser implementada em um projeto. O rastreamento do tempo de desenvolvimento foi possível através do *WakaTime*, que é um *plug-in* para Integrated Development Environment (IDE).

O *WakaTime* possui uma precisão de 1 microssegundo e toda vez que o arquivo é alterado, salvo ou após 2 minutos de digitação ativa no mesmo arquivo um *heartbeats* é enviado, pela IDE, para os servidores do *WakaTime*, e o tempo é informado no perfil do desenvolvedor na plataforma online (WAKATIME, 2013).

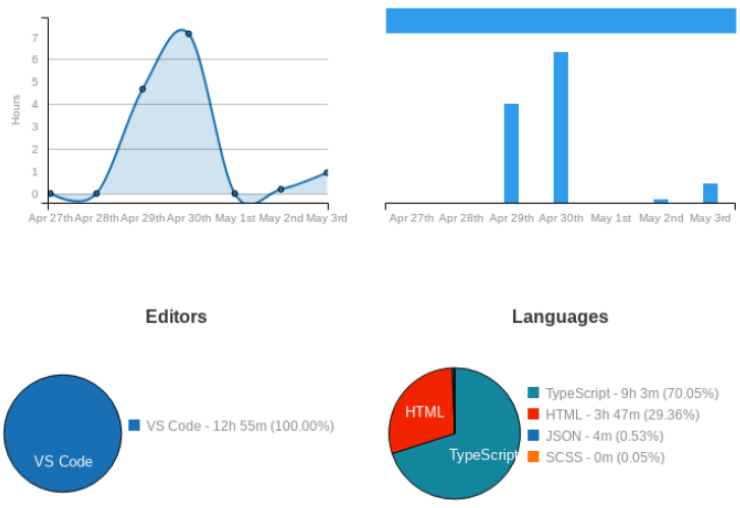
3.2 Projeto Piloto

Esta seção tem como objetivo ilustrar o desenvolvimento das técnicas de implementação de variabilidade voltadas para linguagem de programação. A execução do projeto consiste na implementação de um sistema de exame online com a utilização dos *frameworks Angular* e *Ionic*. Entende-se como exame online a realização de um teste online para medir o conhecimento dos participantes sobre um determinado tópico. Antes, os alunos tinham que se reunir em uma sala de aula ao mesmo tempo para fazer um exame. Com este sistema os alunos podem fazer o exame online, em seu próprio horário e dispositivo, independentemente de onde estejam. Requer apenas um *tablet* ou *smartphone* com conexão à Internet.

Nesse sistema de exames considerado *single system*, os professores podem criar questões escolhendo a área de estudo da questão, o nível de dificuldade e a forma de como o aluno irá responder, seja de escolha única ou múltipla, ou de texto livre; por fim, os professores selecionam e adicionam as questões para criar o exame. Os alunos podem fazer login ou receber uma chave para acessar o sistema e realizar o exame online, vendo os resultados imediatamente depois. Em Anexo apresenta-se o documento de requisitos contendo informações detalhadas e os casos de uso implementados no projeto. É possível visualizar todos os projetos desenvolvidos com e sem utilização das técnicas de implementação de variabilidade no repositório[1] do Github.

O primeiro projeto desenvolvido foi sem o uso de nenhuma técnica de implementação de variabilidade. A Figura 9 apresenta os dados coletado pelo *WakaTime* como o tempo em horas de desenvolvimento, os dias no qual o projeto foi implementado, as IDEs utilizadas e as linguagens programadas. Observa-se que o *WakaTime* mensurou aproximadamente 12h55min de desenvolvimento neste projeto inicial.

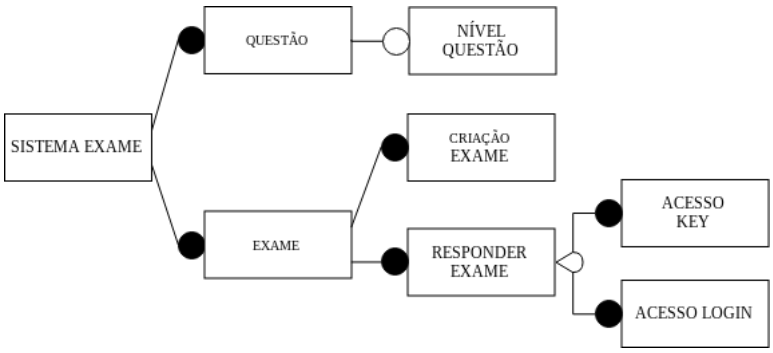
Figura 9 – Dados do projeto sem a utilização de técnica de implementação de variabilidade



[1] https://github.com/taironedias/tcc-feature-oriented-spl.git

O documento de requisitos, em anexo, apresenta casos de uso que contêm *features* alternativa e opcional. Dessa forma, foi criado um modelo de *feature* sobre o projeto piloto, o qual servirá de orientação para a utilização das técnicas de implementação de variabilidade. A Figura 10 apresenta apenas as *features* que possuem característica variável, como a forma do aluno realizar o exame e se a questão pode ou não ter nível de dificuldade.

Figura 10 – Modelo de *feature* do projeto piloto



Percebe-se neste modelo de *features* que se pode obter quatro derivações diferentes do mesmo produto. Ou seja, o sistema 1 pode ter o acesso ao sistema para responder o exame via a chave e não ter o nível da questão; o sistema 2 pode ter o acesso ao sistema via a chave e ter o nível da questão; o sistema 3 o acesso torna-se via usuário e senha e pode não ter o nível da questão; e, o sistema 4 o acesso pode ser via usuário e senha e ter o nível da questão. Esses quatro possíveis sistemas são o que se deseja derivar utilizando as técnicas de implementação de variabilidade.

3.2.1 Parâmetros

O desenvolvimento do projeto piloto utilizando a técnica de implementação de variabilidade parâmetros foi baseada na análise do modelo de *features* (Figura 10) sobre o projeto já implementado sem a utilização de técnicas de implementação de variabilidade. Dessa forma, reimplementou-se o projeto transformando-o em LPS conforme a seleção de *features* desejada no modelo de *features*. Inicialmente, foram criados dois arquivos: (i) um no formato json, denominado config.json, que contém as variáveis para a derivação do produto e (ii) um serviço no formato *TypeScript*, denominado ConfigService.ts, para o tratamento do json em qualquer classe do sistema. O arquivo config.json é acessado pelo app.module do projeto, onde o ConfigService é chamado para instanciá-lo. Pode-se visualizar a implementação do config.json e ConfigService.ts na Figura 11.

O tratamento do arquivo config.json para o projeto piloto ocorreu nos arquivos aluno.page.html e aluno.page.ts para a *feature* alternativa. E nos arquivos criar-questao.page.html e .ts, atualizar-questao.page.html e .ts, e selecionar-questao.page.ts para a *feature* opcional. Na Figura 12 observa-se em: aluno.page.html que as tags div são exibidas conforme a instrução ngIf do *Angular* que avalia a variável booleana accessKey.

Figura 11 – Arquivos config.json e ConfigService.ts implementados

config.json
```
{
    "ACCESS":"login",
    "LEVEL":"false"
}
```

config.ts
```
export const ACCESS_BASE = new InjectionToken<string>('ACCESS_BASE');
export const LEVEL_BASE = new InjectionToken<string>('LEVEL_BASE');

export function ConfigFactory(configService: ConfigService, file: string,
                              property: string) {
    return configService.loadJSON(file)[property];
}

@Injectable()
export class ConfigService {
    public config: any;

    constructor(private http: Http) {}

    loadJSON(filePath) {
        const json = this.loadTextFileAjaxSync(filePath, 'application/json');
        return JSON.parse(json);
    }

    loadTextFileAjaxSync(filePath, mineType) {…
    }
}
```

Essa variável accessKey é atribuída true ou false pelo operador ternário, conforme o que é recuperado no arquivo config.json, tornando a mudança do fluxo em tempo de execução e a variabilidade em tempo de compilação.

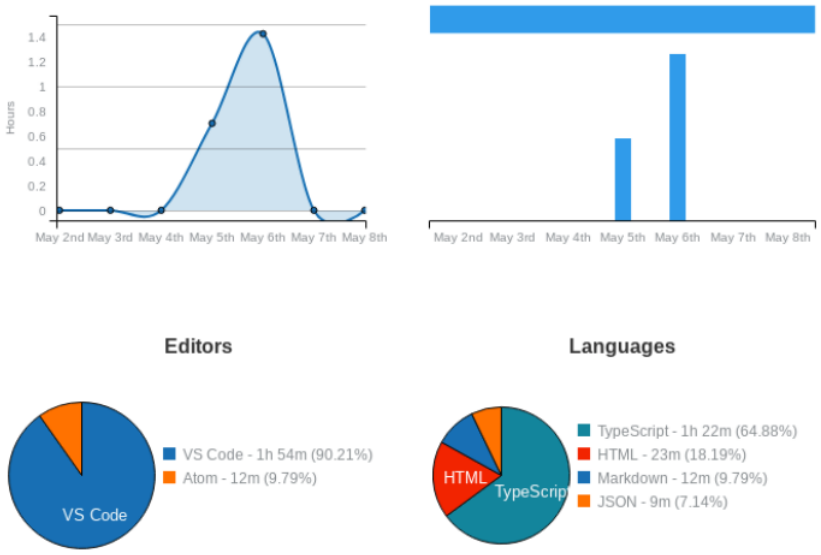
Figura 12 – Arquivos aluno.page.html e aluno.page.ts implementados

aluno.page.html
```
<ion-content padding>

  <div *ngIf="accessKey">…
  </div>

  <div *ngIf="!accessKey">…
  </div>

  <ion-button color="dark" (click)="makeExame()"
    fill="solid" expand="block"> {{ nameButton }} </ion-button>
```

aluno.page.ts
```
  accessKey: boolean;

  constructor(private qstData: QuestionDataService,
              private router: Router,
              private alertCtrl: AlertController,
              private configService: ConfigService,
              @Inject(ACCESS_BASE) accessBase: string) {
    this.accessKey = accessBase === 'key' ? true : false;
  }

  ngOnInit() {
    if (this.qstData.examesArray !== null) {
      this.valuesDefault();
    }
    if (this.accessKey) {
      this.nameButton = 'FAZER EXAME';
    } else {
      this.nameButton = 'LOGIN';
    }
  }
```

A Figura 13 apresenta os dados coletado pelo *WakaTime*. Observa-se que o tempo de desenvolvimento deste projeto durou aproximadamente 2h7min, transformando assim um projeto *single system* em LPS utilizando a técnica parâmetros.

Figura 13 – Dados do projeto utilizando a técnica de implementação de variabilidade parâmetros



3.2.2 Padrões de Projeto

Os padrões de projeto utilizados para implementar a variabilidade no projeto piloto foram o padrão *template-method* e o padrão *strategy*. A escolha desses padrões é justificada pelo fato das técnicas de implementação de variabilidade *frameworks* caixa branca e caixa preta utilizarem também desses padrões como uma parte de suas implementações, vista na Subseção 2.3.3.

3.2.2.1 Padrão *Template-Method*

No padrão *template-method*, o código foi desenvolvido reutilizando a implementação do projeto piloto sem utilização de nenhuma das técnicas de implementação de variabilidade. Com isso, o desenvolvimento divide-se em duas partes: (i) estrutura para a variabilidade utilizando arquivo de configuração e (ii) reimplementação das *features* utilizando o padrão *template-method*.

Na primeira parte, Figura 14, foram criados três arquivos, denominados initConfig.json, config.ts e subject.service.ts, que são respectivamente: o arquivo de configuração contendo as variáveis para a variabilidade das *features* alternativa e opcional; a interface, denominada IConfig, para atribuir os valores do initConfig.json a variáveis em *TypeScript* e o serviço, denominado ConfigSubjectService, para acessar o initConfig.json e retornar os dados no formato Observable do tipo IConfig.

A variável "access" presente no initConfig.json pode receber dois tipos de valores para a *feature* alternativa, o "access-key", Figura 14, e o "login-student". Esses valores são necessários, pois servirão de rota para o direcionamento do sistema. Na mesma pasta do arquivo de configuração do projeto existe o arquivo readme.md que explica como a variabilidade deve acontecer, para cada variável.

Figura 14 – Arquivos initConfig.json, config.ts e subject.service.ts implementados

initConfig.json
```
{
    "access":"access-key",
    "level":false
}
```
config.ts
```
export interface IConfig {
    access: string;
    level: boolean;
}
```
subject.service.ts
```
@Injectable()
export class ConfigSubjectService {

    private url = '/assets/config/initConfig.json';

    constructor(private http: HttpClient) {}

    getInitConfig(): Observable<IConfig> {
        return this.http.get<IConfig>(this.url);
    }
}
```

Posteriormente, os dados foram recuperados através do subscribe (método nativo em Observables) em todos os arquivos *TypeScript* das *features* obrigatórias que necessitam tomar uma decisão a respeito das *features* alternativa e opcional. A Figura 15 exemplifica esse contexto em uma parte do código da classe HomePage. Essa classe direciona para o sistema que o usuário acessará: professor ou aluno. No início do carregamento desta classe, o método ngOnInit recupera o valor associado à variável "access" e caso a escolha seja o aluno, a classe HomePage redirecionará para a página de acesso definida no arquivo de configuração.

Figura 15 – Código de recuperação dos dados na classe HomePage

home.page.ts
```
export class HomePage implements OnInit {

  private accessPage: string;

  constructor(public router: Router,
              private configS: ConfigSubjectService) {}

  ngOnInit() {
    this.configS.getInitConfig()
      .subscribe(data => this.accessPage = data.access);
  }

  teacherUser() {…
  }

  studantUser() {
    this.router.navigate([this.accessPage]);
  }
}
```

Na segunda parte, algumas *features* foram reimplementadas utilizando as regras do padrão *template-method* como classes e métodos abstratos, herança e polimorfismo. Os arquivos modificados correspondem a *features* que se conectam com as *features* alternativa e opcional previstas no modelo de *features*. A Figura 16 ilustra a implementação da classe abstrata AccessPage e as declarações dos métodos abstratos letsGo e resetValues, bem como, a herança realizada pela classe concreta AccessKeyPage, tendo que desenvolver os métodos definidos na classe abstrata. Observe que o templateURL das duas classes, ambas estão direcionado para o mesmo arquivo, ou seja, foi criado um único arquivo HTML e utilizado em diferentes classes. De forma, análoga a reimplementação das classes e arquivos HTML aconteceram para as *features* de login do aluno, criar e atualizar questão.

Figura 16 – Implementação da classe abstrata AccessPage e classe concreta AccessKeyPage

access.page.ts

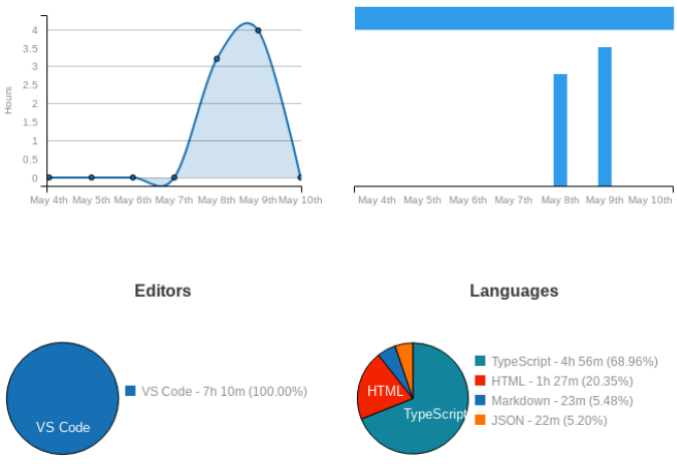
```
@Component({
  selector: 'app-access',
  templateUrl: './access.page.html',
  styleUrls: ['./access.page.scss'],
})
export abstract class AccessPage
  implements OnInit {

  protected title: string;
  protected labelText1: string;
  protected labelText2: string;
  protected label1: string;
  protected label2: string;
  protected typeLabel2: string;
  protected buttonText: string;

  constructor(protected qstData: QuestionDataService,
              private alertCtrl: AlertController,
              protected router: Router) { }

  ngOnInit() { }

  abstract letsGo();
  abstract resetValues();
```

access-key.page.ts

```
@Component({
  selector: 'app-access-key',
  templateUrl: '../access/access.page.html',
  styleUrls: ['./access-key.page.scss'],
})
export class AccessKeyPage extends AccessPage
  implements OnInit {

  ID = 0;
  flag = false;

  ngOnInit() {
    this.valuesDefault();
    this.title = 'Aluno';
    this.labelText1 = 'Digite seu nome:';
    this.labelText2 = 'Digite a chave de acesso:';
    this.typeLabel2 = 'text';
    this.buttonText = 'Fazer exame';
  }

  letsGo() {…
  }

  resetValues() {…
  }
```

A Figura 17 apresenta os dados coletado pelo *WakaTime*. Observa-se que aproximadamente 7h10min de desenvolvimento foram necessárias para transformar um projeto *single system* em variável, utilizando a técnica padrão *template-method*.

Figura 17 – Dados do projeto utilizando a técnica de implementação de variabilidade padrão *template-method*



3.2.2.2 Padrão *Strategy*

Assim como no padrão *template-method*, o projeto piloto com o padrão *strategy* foi desenvolvido reutilizando os códigos do projeto *single system*. Dividiu-se essa implementação em duas etapas: (i) estrutura para a variabilidade utilizando arquivo de configuração e (ii) reimplementação das *features* utilizando o padrão *strategy*.

A primeira etapa assemelha-se ao visualizado no padrão *template-method* na Figura 14, onde se apresentam três arquivos: initConfig.json que é o arquivo de configuração; a interface IConfig e o serviço ConfigSubjectService para acessar o arquivo de configuração e retornar os dados das *features* alternativa e opcional no formato Observable do tipo IConfig. A diferença dessa etapa no padrão *strategy* para o padrão *template-method* está nos valores passados para a variável access no initConfig.json. Se antes, os possíveis valores a serem passados para a variável access eram "access-key" e "login-student", agora se têm o "AccessWithKeyService" e "AccessWithLoginService" que são nomes de serviços.

Recuperaram-se os dados da mesma forma que o apresentado no padrão *template-method*, através do método subscribe. Na classe HomePage os dados recuperados referentes ao modo de acesso, provenientes do arquivo de configuração, foram passados como parâmetro de rota para a página access. Em AccessPage, Figura 18, resgatou-se o valor recebido como parâmetro da rota e o método getServiceByName foi chamado para converter a string passada em um serviço, como foi definido no arquivo de configuração.

Figura 18 – Conversão string em *service* na classe AccessPage

access.page.ts

```
15  export class AccessPage implements OnInit {
16    :
17    :
27    ngOnInit() {
28      const data = this.route.snapshot.paramMap.get('data');
29      this.chosenAccess = this.getServiceByName(data);
30      this.valuesDefault();
31    }
32
33    getServiceByName(nameService: string) {
34      return this.injector.get(AccessStrategyToken)
35        .find(x => x.constructor.name === nameService);
36    }
```

Na segunda etapa, a reimplementação ocorreu de acordo com as orientações do método *strategy*, ou seja, criou-se a interface IAccessStrategy que contém as assinaturas de variáveis e métodos, a classe contexto que é a AccessPage e as classes que implementam a interface AccessWithLoginService e AccessWithKeyService, como se pode visualizar na Figura 19.

Figura 19 – Interface IAccessStrategy e classe AccessWithKeyService implementadas

iacesso.ts

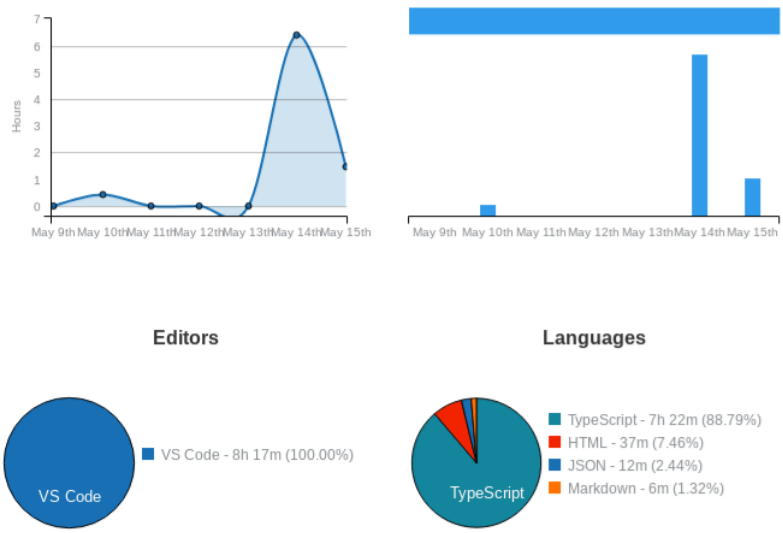
```
3  export interface IAcessoStrategy {
4      title: string;
5      labelText1: string;
6      labelText2: string;
7      label1: string;
8      label2: string;
9      typeLabel2: string;
10     buttonText: string;
11     callback: string;
12
13     makeAction(qstData: QuestionDataService): any[];
14 }
15
```

access-with-key.service.ts

```
5  @Injectable()
6  export class AccessWithKeyService implements IAcessoStrategy {
7
8      title = 'Aluno';
9      labelText1 = 'Digite seu nome:';
10     labelText2 = 'Digite a chave de acesso:';
11     typeLabel2 = 'text';
12     label1: string;
13     label2: string;
14     buttonText = 'Fazer exame';
15     callback: string;
16     ID = 0;
17     flag = false;
18
19     makeAction(qstData: QuestionDataService): any[] {…
41     }
42
43 }
```

Os dados coletados pelo *WakaTime* são apresentados na Figura 20. O tempo de desenvolvimento durou aproximadamente 8h17min para a implementação da técnica de variabilidade padrão *strategy* sobre o projeto *single system*.

Figura 20 – Dados do projeto utilizando a técnica de implementação de variabilidade padrão *strategy*



3.2.3 *Frameworks*

De acordo com Apel *et al.* (2013), ao utilizar *frameworks* o software é derivado automaticamente conforme a seleção de *plug-ins*, que representam as *features*, implementados pelo padrão *template-method* ou *strategy*. Diante desse contexto, para utilizar *frameworks* tanto de caixa branca quanto de caixa preta como técnica de implementação de variabilidade em LPS as tecnologias *Angular* e *Ionic* deveriam fornecer um comando no CLI para a geração automática do produto e suas variações, caso necessário. Até o momento da pesquisa deste trabalho tanto a versão 8.0 do *Angular* quanto a versão 4.0 do *Ionic* não suportavam essa automação na composição automática do software, tornando

impossível uma apresentação prática dessa técnica de implementação de variabilidade sobre o projeto piloto.

3.2.4 Componentes e Serviços

Por padrão ao criar um projeto em *Angular*, ou *Ionic*, nativamente têm-se o conceito de componentes e serviços. Essas tecnologias possuem comandos no CLI que geram automaticamente arquivos, realizando as ligações necessárias nos arquivos de configuração e módulo existentes no projeto provocando uma redução no tempo de desenvolvimento, caso feito manualmente.

Para *Ionic*, esses comandos são *ionic generate service*, *ionic generate component* ou *ionic generate page*. Tanto o *component* quanto *page* são comandos que criam uma pasta dentro do projeto, contendo os arquivos HTML, *TypeScript* e CSS. A diferença é que o *page* atualiza o arquivo app-routing.module.ts adicionando um novo caminho de rotas. Caso um *component* criado necessite ser chamado como rota de navegação no sistema, deve-se criar manualmente um arquivo *.module.ts dentro da pasta do *component* e adicionar o nome desse arquivo como caminho do app-routing.module.ts.

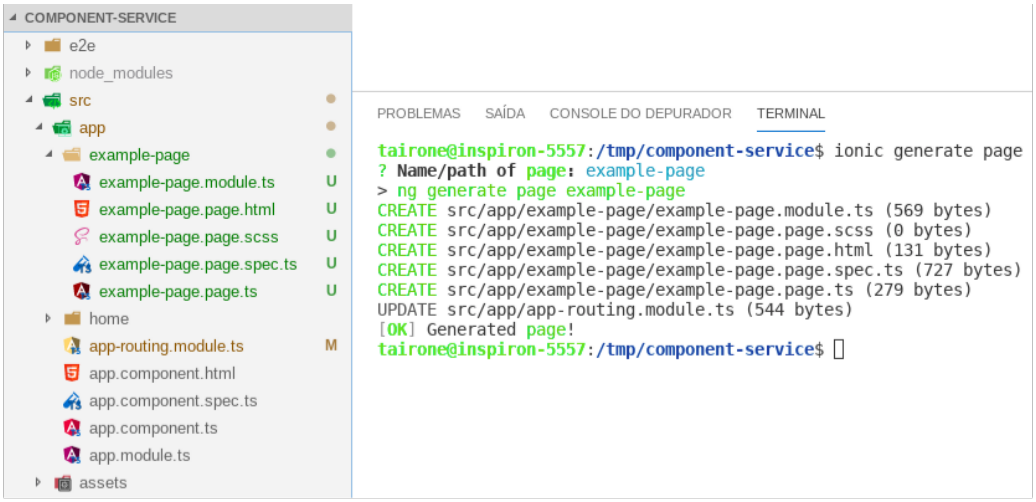
Na prática, a diferença básica entre os comandos no *Ionic*, é que o *page* pode ser executado como se fosse uma aplicação independente do projeto, já o *component* necessita de algum *page* para ser exibida. A Figura 21 exemplifica a utilização dos comandos tanto para gerar uma página quanto um componente. Percebe-se que na Figura 21(a) ao utilizar o comando para criar uma página, o CLI solicita um nome para a página e, após informar, é gerado um registro das ações realizadas. Ao final, foram criados cinco arquivos e o arquivo app-routing.module.ts foi atualizado. Já na Figura 21(b) o mesmo comportamento acontece para o CLI: um nome é solicitado para o componente e o registro de ações é exibida. No entanto, nesse caso apenas quatro arquivos foram criados e nenhuma atualização nos arquivos existente foi realizada.

Os serviços criados pelo CLI tornam-se globais para o projeto se adicionados no vetor de *providers* no arquivo app.module. A utilização de serviços no projeto piloto, começou desde o início da pesquisa ao implementar no projeto *single system*, por conta de regras de negócio da linguagem.

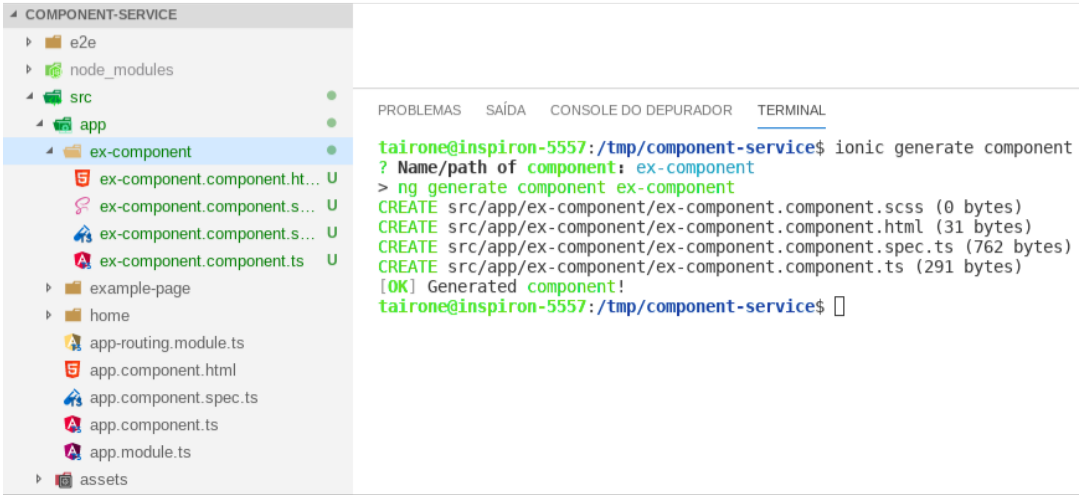
Para essa técnica de implementação de variabilidade os códigos do projeto *single system* foram reutilizados. Como mencionado no início desta seção, o *Ionic* contém nativamente a criação de *component* e *page*, então, boa parte do código para essa técnica foi implementada sem conhecimento no projeto *single system*. Porém, (i) foram criados os arquivos de configuração semelhante ao visualizado na Figura 14 para que a variabilidade das *features* alternativa e opcional possam ocorrer em um mesmo arquivo, (ii) as formas de acesso foram separadas para responder ao exame em duas *pages* e (iii) um *component*

foi adicionado para a *feature* opcional.

Figura 21 – Comandos no *Ionic* CLI



(a)



(b)

Para a *feature* alternativa presente no modelo de *features*, a implementação foi a mesma visualizada na Figura 15, onde se recuperou a forma de acesso oriunda do arquivo de configuração e foi definida como rota. Ou seja, o valor presente na variável do initConfig.json deve conter o mesmo nome das possíveis rotas de navegação para essa variabilidade, que são: “login-student” e “access-key”.

A Figura 22 apresenta a implementação da variabilidade para a *feature* opcional. Um *component* no projeto foi adicionado e declarado o seu nome no módulo da *feature* criar-questao. Assim, o *component* foi utilizado dentro de CriarQuestaoPage, ao inserir a *tag* do *selector* do *component* no criar-questao.page.html e recuperar o seu valor através do conceito *Component Interaction*.

Os dados coletados pelo *WakaTime* são apresentados na Figura 23. Observa-se que aproximadamente 1h54min foi o tempo necessário para reorganizar e implementar

Figura 22 – Utilizando *Component* em *Page* no *Ionic*

criar-questao.page.html

```
116   <!-- NIVEL DE DIFICULDADE -->
117   <div *ngIf="level">
118     <app-nivel></app-nivel>
119   </div>
```

nivel.component.html

```
1   <ion-item>
2     <ion-label>Nível de dificuldade:</ion-label>
3     <ion-select interface="popover" placeholder="---"
4     [(ngModel)]="nivel">
5       <ion-select-option *ngFor="let item of niveis">
6         {{ item }}
7       </ion-select-option>
8     </ion-select>
9   </ion-item>
```

nivel.component.ts

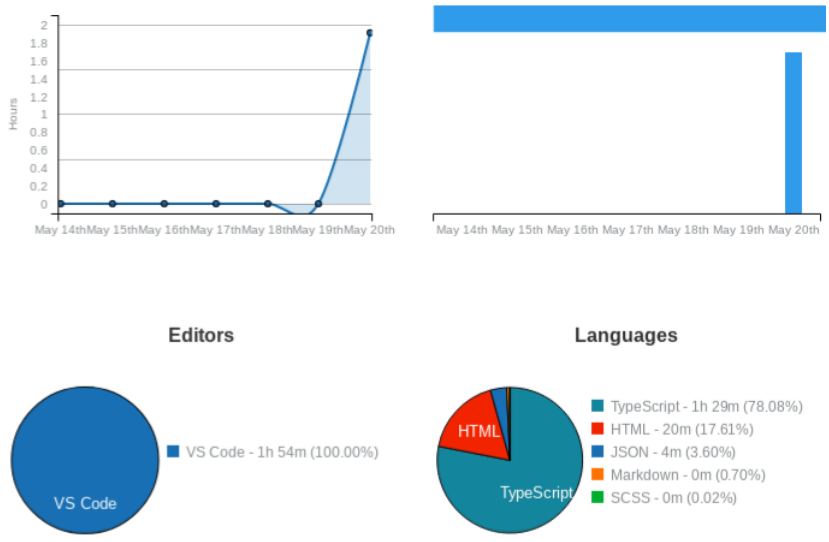
```
8   export class NivelComponent implements OnInit {
9
10    niveis = ['1', '2', '3'];
11    nivel: string;
12
13    constructor() { }
```

criar-questao.page.ts

```
45    level: boolean;
46    @ViewChild(NivelComponent) nivelComponent;
47    receberNivel: string;
48
49    ngOnInit() {
50      this.valuesDefault();
51      this.configS.getInitConfig()
52        .subscribe(data => this.level = data.level);
53    }
```

as *features* alternativa e opcional sobre o projeto *single system*. Apesar do *Angular* e *Ionic* possuírem um comando no CLI para a composição automática de um *component*, a maior parte desse tempo corresponde à implementação da *feature* opcional, pois houve uma integração e comunicação entre *component* e *page* de forma manual, referente aos requisitos da *feature*.

Figura 23 – Dados do projeto utilizando a técnica de implementação de variabilidade componentes



Com a implementação das técnicas de variabilidade no projeto piloto em ambiente controlado, foi possível identificar a aplicabilidade das técnicas sobre os *frameworks* JS modernos, *Angular* e *Ionic*. O próximo capítulo apresenta um estudo de caso sobre um projeto real onde se investigaram os benefícios, deficiências e viabilidades das técnica dentro de um contexto. O protocolo do estudo de caso é explicado, permitindo um melhor entendimento do cenário e para que outras pesquisas possam replicá-lo.

4

Estudo de Caso

Neste capítulo será apresentada uma pesquisa de estudo de caso com o objetivo de investigar as técnicas de implementação de variabilidade em um contexto real. Segundo Merriam (2009 apud VALE, 2012), o estudo de caso caracteriza uma compreensão sendo o pesquisador o principal instrumento para a coleta e análise do estudo de pequisa, tornando o produto final altamente descritivo mesmo com uma estratégia de investigação indutiva.

Assim, investigar-se-ão as técnicas de implementação de variabilidade em LPS para um sistema de configuração de roteadores com seu respectivo conjunto de *features*. Inicialmente, apresentar-se-á o protocolo de estudo de caso que explicará as definições em torno da pesquisa e em sequência os resultados e conclusões obtidos.

4.1 Protocolo do Estudo de Caso

Nas próximas subseções apresentar-se-ão as definições do protocolo incluindo (i) objetivo, (ii) o sistema ou caso delimitado, (iii) unidades de análise, (iv) questões de pesquisa do estudo de caso e (v) procedimento de análise de dados, baseado nas diretrizes de Runeson e Höst (2009 apud VALE, 2012).

4.1.1 Objetivo

O objetivo da aplicação deste estudo de caso é investigar quais técnicas de implementação de variabilidade são aplicáveis, dado um conjunto de *features* obrigatórias, alternativas e opcionais de um projeto de mundo real baseado em *frameworks* JS modernos visando identificar os benefícios, deficiências e viabilidades das técnicas analisadas sobre os critérios de avaliação.

4.1.2 O Caso

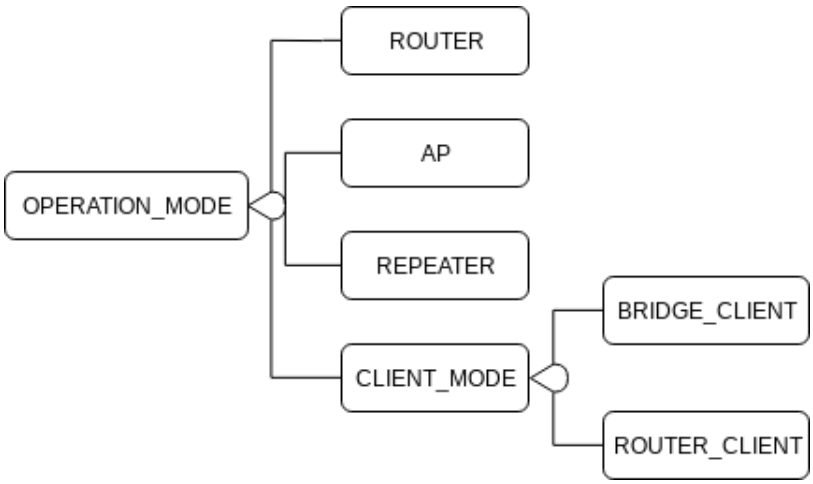
O caso refere-se ao projeto de uma empresa que fabrica produtos e soluções em segurança, redes e telecomunicações para pequenas e médias empresas com aproximada-

mente 2.261 funcionários. Dentro de uma das unidades de negócios da empresa, têm-se a fabricação de modems, roteadores, adaptadores *wireless*, placas de rede e *switches*, sendo as unidades fabris instaladas em três estados brasileiros.

Um de seus produtos de fabricação são roteadores no qual a empresa também fornece o software de instalação e configuração. Até o momento desta pesquisa, a empresa fabricava 12 modelos de roteadores que possuíam seus respectivos software de configuração. Percebendo a dificuldade de produzir e manter esses software a empresa optou por transformar o desenvolvimento utilizando os conceitos de LPS, visando aproveitar as vantagens e benefícios dessa técnica.

Inicialmente, a equipe de desenvolvimento de sistemas da empresa adotou o *framework JS Ionic*, pela capacidade de derivação para as plataformas *Android* e *iOS*, e pela familiaridade da equipe por ser programada em uma linguagem web. A equipe também realizou um estudo para verificar quais *features* estão presentes em cada produto, suas semelhanças e variações, produzindo, ao final, um modelo de *features*. A Figura 24 apresenta o modelo de *feature*, para um módulo do projeto.

Figura 24 – Modelo de *Features* do estudo de caso



O modelo de *features*, apresentado na Figura 24, é sobre a perspectiva da variabilidade do sistema em tempo de execução. O módulo no qual o modelo de *features* está representando é denominado *"Network"*, cuja responsabilidade é configurar o modo de operação do roteador. Diante dos modelos produzidos pela empresa, o dispositivo possui a seguinte variabilidade: *Router*, *AP*, *Repeater* e modo *Client*. A escolha desse módulo para a implementação das técnicas de variabilidade é justificada por ser uma funcionalidade principal do sistema e por abranger todas as classificações de *features*, ou seja, *features* obrigatórias, alternativas e opcionais.

A Seção 4.2 apresenta como as técnicas de implementação de variabilidade foram aplicadas e a investigação realizada sobre os benefícios e deficiências de cada técnica sob os critérios de avaliação. Pelo termo de sigilo acordado, pequenos trechos de códigos, para

exemplificação, serão exibidos de forma que a empresa permaneça no anonimato.

4.1.3 Unidades de Análise

As unidades de análise deste estudo de caso referem-se às técnicas de implementação de variabilidade sobre a tecnologia *language-based*, conforme justificada no início do Capítulo 3, tendo a primeira técnica parâmetros que utilizam estruturas condicionais para implementar variabilidades. As técnicas presentes nos padrões de projeto, nas quais serão usadas apenas os padrões *template-method* e *strategy*, utilizam os conceitos de classes abstratas e interfaces, respectivamente. E as técnicas componentes e serviços, que abordam conceitos de adicionar arquivos ou trechos de códigos pré-definidos ao projeto e assim complementá-lo. A técnica *framework* não será abordada tendo em vista que não foi possível implementá-la no projeto piloto por uma limitação do *framework* JS.

4.1.4 Questões de Pesquisa

As questões de pesquisa avaliam o caso sobre os benefícios, deficiências e viabilidades das técnicas de implementação de variabilidade sobre a perspectiva do pesquisador deste trabalho. Dessa forma, as seguinte questões foram definidas:

1. **As técnicas de implementação de variabilidade são aplicáveis no contexto do estudo de caso?**
   Justificativa: essa questão verifica se todas as técnicas implementadas no projeto piloto, que foi em um ambiente controlado, podem ser aplicadas em um projeto real, onde existem variáveis e contextos incontroláveis pelo pesquisador.

2. **Quais benefícios as técnicas implementadas em *frameworks* JS modernos trazem para o desenvolvimento do projeto em LPS?**
   Justificativa: nessa questão, o pesquisador verifica quais benefícios foram diagnosticados para cada técnica sobre os critérios de avaliação.

3. **Quais deficiências encontradas em cada técnica de implementação de variabilidade?**
   Justificativa: o pesquisador verifica quais deficiências ou limitações foram encontradas para cada técnica aplicada ao estudo de caso, sobre os critérios de avaliação.

4. **Qual a viabilidade das técnicas de implementação de variabilidade ao serem aplicadas no projeto?**
   Justificativa: nessa questão, o pesquisador verifica a praticabilidade das técnicas em um projeto de contexto real.

4.1.5 Análise de Dados

A estratégia para analisar os dados foi feita por uma identificação textual de cada critérios de avaliação para cada técnica de implementação de variabilidade desenvolvida, tendo como referência o que foi apresentado na Seção 3.1. Ao final, apresenta-se uma tabela de critérios de avaliação e técnicas de implementação de variabilidade em LPS sobre o projeto do estudo de caso.

Foi realizado também um comparativo entre os resultados encontrados na literatura de Apel *et al.* (2013) apresentados na Tabela 2 e os resultados dessa nova tabela, ao aplicar as técnicas de implementação de variabilidade em um contexto real.

4.2 Resultados

4.2.1 Parâmetros

O roteador contém uma API no qual o sistema se comunica e recebe informações para a configuração correta do dispositivo. Fazendo uma analogia, essa API possui um comportamento semelhante ao arquivo de configuração que se desenvolveu no projeto piloto, ou seja, contém informações que se referem às *features* opcionais *Client_Mode* e *Repeater*. A classe principal de configuração da rede no sistema é a "NetworkHomePage" que incorpora o componente OperationModeSelectComponent. As informações das *features* opcionais são armazenadas nas variáveis enabled_wireless_client_mode e enabled_wireless_repeater_mode, respectivamente.

Na Figura 25 o componente OperationModeSelectComponent consulta a classe OperationModeData passando como parâmetros valores booleanos presentes nas variáveis enabled_wireless_client_mode e enabled_wireless_repeater_mode. Por padrão, a classe OperationModeData cria uma lista de dicionários contendo os dados das *features* obrigatórias *Router* e *AP*. As *features* opcionais *Client_Mode* e *Repeater* são adicionadas na lista se os parâmetros recebidos forem verdadeiros, ao final a lista é retornada. Uma vez que a OperationModeSelectComponent obtém essa lista, a interface é montada pelas instruções do *Angular* ngSwitch e ngIf, e o fluxo de execução do código, para a configuração do roteador, é controlado por instruções condicionais.

Com base nos critérios de avaliação e a implementação realizada neste projeto utilizando a técnica parâmetros, apresentam-se os seguintes benefícios e deficiências identificadas:

- ❑ O tempo de desenvolvimento foi aproximadamente 15min;
- ❑ A granularidade é refinada por conta das instruções condicionais aplicarem a variabilidade;

Figura 25 – Classes OperationModeSelectComponent e OperationModeData

operation-mode-select.ts

```
61      /*
62       * class OperationModeSelectComponent
63       */
64      private defineOperationModes() {
65          const datas = OperationModeData.getDatasAsJson(
66              this.enabled_wireless_client_mode,
67              this.enabled_wireless_repeater_mode
68          );
```

operation-mode-data.ts

```
11  export class OperationModeData {
12      static getDatasAsJson(clientModeEnabled: boolean, repeaterModeEnabled: boolean) {
13
14          let deviceModes = [
15              this.getRouterDataAsJson(),
16              {
17                  value: OperationModeType.AP,
18                  title: 'NETWORK.OPERATION_MODE.AP',
19                  tabs: [OperationModeTab.WIFI, OperationModeTab.LAN]
20              }
21          ];
22
23          if (clientModeEnabled) {
24              deviceModes = deviceModes.concat([
25                  {…
29                  },
30                  {…
34                  }
35              ]);
36          }
```

- ❏ A técnica parâmetros é parcialmente uniforme, pois foi possível implementar a técnica em *Angular* e *Ionic*, com a linguagem de programação *TypeScript*, porém não foi aplicável a artefatos *noncode*;
- ❏ Houve um baixo planejamento prévio, pois não necessitou planejar para implementar a variabilidade com instruções condicionais, sendo esse critério atendido.
- ❏ A rastreabilidade foi possível em todas as *features*, utilizando convenção de nomenclatura em variáveis a serem analisadas nas instruções condicionais;
- ❏ Com um desenvolvimento modular, adotado pela equipe do projeto, houve uma boa separação de *concerns*, no entanto, o *information hiding* não foi possível obter por limitação da técnica de implementação parâmetros.

4.2.2 Padrões de Projeto

4.2.2.1 Padrão *Template-method*

No padrão *template-method* inicialmente tentou-se desenvolver utilizando a mesma lógica no projeto piloto, receber da API as *features* que devem ser adicionadas e utilizar como rota de páginas. No entanto, a estrutura de rotas adotada pela equipe do projeto do estudo de caso impossibilita que isso seja feito. O problema surge por conta da quantidade de router-outlets utilizadas no projeto pela equipe de desenvolvimento e pela integração de componentes às páginas. Seria possível implementar a mesma estrutura utilizada no

projeto piloto se houvesse uma mudança por completo na estrutura de rotas e nas regras de negócio do projeto. Entretanto, isso acabaria caracterizando-se como um projeto em ambiente controlado, logo, os objetivos deste estudo de caso não seriam cumpridos.

Assim, foi abordada outra estratégia para utilizar o padrão *template-method* no projeto. Cada modo de operação, contém dados que representam os tipos de configuração: Wi-Fi, WAN e LAN. Analisando os tipos de configuração de cada modo de operação, detectou-se que a configuração de Wi-Fi e LAN estão presentes em todos os modos e a configuração WAN apenas nos modos de operação de router.

Dessa forma, aplicou-se o padrão *template-method* para a montagem dos modos de operação do roteador. Criou-se a classe abstrata OperationModeTypeClass que define as propriedades "value" (string com nome do modo de operação a ser analisada pelo ngSwitch), "title" (string contendo o nome da operação para ser exibida no ion-select) e "tabs" (lista de string para montar as abas dos tipo de configuração), sendo esta última já inicializada com os valores padrões identificados. Além das propriedades criadas, há também a assinatura do método "getDeviceMode". Para cada modo de operação apresentado na Figura 24 criou-se uma classe concreta que herda de OperationModeTypeClass.

Inicialmente o fluxo de execução é o mesmo da técnica parâmetros. Ou seja, a classe principal de configuração da rede é a "NetworkHomePage" que incorpora o componente OperationModeSelectComponent. Esse componente realiza uma chamada na classe OperationModeData passando como argumento do método se o roteador possui ou não as *features* opcionais.

Como se percebe na Figura 26, a classe OperationModeData cria uma lista de modos de operação instanciando as classes padrões OperationModeTypeRouter e OperationModeTypeAP. Caso algum parâmetro seja verdadeiro, a condição do if será satisfeita e a inclusão de novos modos de operação é realizada. Percebe-se que a classe OperationModeTypeClient foi instanciada duas vezes, pois o modo *Client_Mode* possui internamente dois modos de operação: *Bridge Client* e *Router Client*. Em ambas as instâncias o valor e título foram informados, no entanto, um dos modos possui o tipo de configuração WAN.

Assim, para criar corretamente os tipos de configuração para cada modo, utilizou-se do conceito de parâmetro opcional, Figura 27. Desenvolveu-se internamente na classe mais um método denominado "combineDeviceModes", para combinar esses dois modos de operação. Por fim, o método "getDataAsJson" presente na classe OperationModeData retorna para o componente OperationModeSelectComponent a lista com todos os modos de operação possíveis para aquele roteador.

Figura 26 – Implementação da classe OperationModeData

operation-mode-data.ts

```
15  export class OperationModeData {
16      static getDatasAsJson(clientModeEnabled: boolean, repeaterModeEnabled: boolean) {
17          let deviceModes = [
18              new OperationModeTypeRouter('wan').getDeviceMode(),
19              new OperationModeTypeAP().getDeviceMode()
20          ];
21
22          if (clientModeEnabled) {
23              deviceModes = deviceModes.concat(
24                  new OperationModeTypeClient(OperationModeType.ROUTER_CLIENT,
25                      'NETWORK.OPERATION_MODE.ROUTER_CLIENT', 'wan')
26                      .combineDeviceModes(
27                          new OperationModeTypeClient(OperationModeType.BRIDGE_CLIENT,
28                              'NETWORK.OPERATION_MODE.BRIDGE_CLIENT').getDeviceMode()
29                      )
30              );
31          }
```

Figura 27 – Implementação do padrão *template-method* para o modo de operação *Client_Mode*

operation-mode-type.ts

```
1  export abstract class OperationModeTypeClass {
2      protected value: string;
3      protected title: string;
4      protected tabs = ['wifi', 'lan'];
5
6      abstract getDeviceMode();
7  }
```

operation-mode-type-client.ts

```
3  export class OperationModeTypeClient extends OperationModeTypeClass {
4
5      constructor(value: string, title: string, tab?: string) {
6          super();
7          this.value = value;
8          this.title = title;
9          if (tab !== undefined) {
10             this.tabs.push(tab);
11         }
12     }
13
14     getDeviceMode() {
15         return {
16             value: this.value,
17             title: this.title,
18             tabs: this.tabs
19         };
20     }
21
22     combineDeviceModes(data) {…
31     }
```

4.2.2.2 Padrão *Strategy*

Semelhante ao executado no padrão *template-method*, o padrão *strategy* utilizou os conceitos de interface para implementar a variabilidade na composição das abas de configuração para cada modo de operação.

Inicialmente criou-se a interface, semelhante à classe abstrata, porém sem atribuições; apenas a assinatura do método e das variáveis. Para cada modo de operação previsto no modelo de *features* criaram-se classes concretas que implementam a interface, atribuindo valores às variáveis e desenvolvendo o método. Para que cada uma das classes seja chamada e, assim, criar a lista de modo de operações com suas respectivas configurações, criou-se um token e referenciou-o para cada classe concreta no network.module em providers. Nesse

processo, pode-se fazer uma analogia com o desenvolvimento do padrão *strategy* no projeto piloto.

Após recuperar os dados oriundos da API do roteador, o método "defineOperationModes" da classe OperationModeSelectComponent cria internamente uma lista com os modos de operação e seus respectivos tipos de configurações (Wi-Fi, WAN, LAN), como se pode visualizar na Figura 28. O método "getClassByName" retorna a classe conforme o nome passado como parâmetro. Por fim, o método "getDeviceMode" retorna um dicionário contendo as informações necessárias para montar a interface e direcionar o fluxo de execução para cada modo de operação e seus respectivos tipos de configuração.

Figura 28 – Implementação do padrão *strategy* para os modos de operação

operation-mode-select.ts

```
64    /*class OperationModeSelectComponent */
65    private defineOperationModes() {
66
67        const datas = [
68            this.getClassByName('OperationModeTypeRouter').getDeviceMode(),
69            this.getClassByName('OperationModeTypeAP').getDeviceMode()
70        ];
71
72        if (this.enabled_client_mode) {
73            datas.push(this.getClassByName('OperationModeTypeBridgeClient')
74                .getDeviceMode());
75            datas.push(this.getClassByName('OperationModeTypeRouterClient')
76                .getDeviceMode());
77        }
78
79        if (this.enabled_repeat_mode) {
80            datas.push(this.getClassByName('OperationModeTypeRepeater')
81                .getDeviceMode());
82        }
```

Com base nos critérios de avaliação e a implementação realizada neste projeto utilizando padrões de projeto sobre as técnicas padrão *template-method* e *strategy*, apresentam-se os seguintes benefícios e deficiências identificadas:

- ❏ O tempo de desenvolvimento médio das técnicas foi de aproximadamente 1h56min. No entanto, o tempo médio gasto para tentar implementar cada técnica conforme a orientação da literatura foi aproximadamente 5h11min, o que reflete em um alto planejamento prévio;
- ❏ A granularidade é grossa, uma vez que as *features* são implementadas ao nível de classes e interfaces;
- ❏ De modo geral, padrões de projeto é considerado parcialmente uniforme, pois foi possível implementar a técnica em *Ionic*, com a linguagem de programação *TypeScript*, porém não foi aplicável a artefatos *noncode*;

- ❏ Como citado acima, houve um alto planejamento prévio, pois identificar e discernir quais *features* podem ser combinadas e como organizá-las com base nos padrões sobre um projeto de estudo de caso, não é simples comparado à técnica parâmetros, por exemplo;
- ❏ A rastreabilidade foi evidente, pois se trata de classes e interfaces que fizeram uso de nome das *features* no documento de requisitos;
- ❏ O projeto do estudo de caso foi desenvolvido de forma modular, que por consequência influência na separação de *concerns*. O *information hiding* surge com o uso de classes abstratas e interfaces contendo apenas as assinaturas de métodos e variáveis.

4.2.3 Componentes e Serviços

Como citado na Subseção 3.2.4, tanto o *Ionic* quanto *Angular* incentivam um desenvolvimento a base de componentes e serviços, fornecendo documentações e instruções para geração automática dos mesmos no projeto. Intencionalmente ou não, a equipe de desenvolvimento da empresa, implementou o projeto do estudo de caso utilizando essa técnica de variabilidade em LPS.

Dessa forma, reimplementou-se para mensurar alguns critérios de avaliação utilizando componentes e serviços, e apresentar os seguintes benefícios e deficiências identificadas:

- ❏ O tempo neste caso para gerar componentes ou serviços pelos comandos do CLI seria em média 20s, mas como só a geração desses arquivos não é suficiente para implementar a variabilidade, adotou-se a técnica parâmetros totalizando um tempo de 15min;
- ❏ A granularidade é grossa, uma vez que as *features* são identificadas ao nível de pacotes;
- ❏ Apesar da literatura informar o contrário, foi considerado parcialmente uniforme. Foi possível implementar a técnica em *Ionic*, com a linguagem de programação *TypeScript*, porém não foi aplicável a artefatos *noncode*;
- ❏ O pré-planejamento resumiu-se em modularizar o sistema, então, comparado ao padrões de projeto houve um baixo planejamento prévio;
- ❏ A rastreabilidade foi evidente, pois se trata de pacotes que fizeram uso de nome das *features* no documento de requisitos;
- ❏ Houve separação de *concerns*, por ser nativamente modular. A técnica em si, por conta dos comandos do CLI, não oculta suas informações através de uma interface, dessa forma não foi possível obter o *information hiding*.

4.3 Discussão

Finalizando o desenvolvimento das técnica de implementação de variabilidade sobre o projeto do estudo de caso, qualificaram-se os benefícios e deficiências identificadas para cada critério de avaliação e técnica de implementação de variabilidade, ao utilizar os *frameworks* JS modernos, apresentado na Tabela 3. Adicionou-se também o tempo de desenvolvimento como critério de avaliação.

Calculou-se a média do tempo total de desenvolvimento das técnica implementadas e classificou-se como baixo para um tempo inferior ou igual à 48 minutos e alto para um tempo superior à 48 minutos. Esse tempo total corresponde apenas ao tempo de desenvolvimento da técnica sobre o projeto do estudo de caso.

Tabela 3 – Características das técnicas de implementação de variabilidade quanto aos critérios de avaliação em LPS sobre o projeto do estudo de caso

	Parâmetros	**Padrões de Projeto**	**Componentes e Serviços**
Pré-planejamento	baixo	alto	baixo
Rastreabilidade	possível	possível	possível
Separação de *concerns*	possível	nativo	nativo
Information hiding	impossível	nativo	impossível
Granularidade	fina	grossa	grossa
Uniformidade	parcial	parcial	parcial
Tempo de desenvolvimento	baixo	alto	baixo

Realizando uma comparação com a Tabela 2, percebe-se que alguns critérios de avaliação não foram equivalentes ao apresentado por Apel *et al.* (2013), como (i) o esforço de pré-planejamento de componentes e serviços que na literatura apresenta um alto planejamento, enquanto que na prática, por conta dos *frameworks Angular* e *Ionic*, possui um baixo planejamento; (ii) a rastreabilidade que em parâmetros é difícil obtê-la, mas no estudo de caso foi possível por conta do uso dos nomes das *features* como variáveis nas instruções condicionais; (iii) a separação de *concerns* para parâmetros na literatura é difícil ser realizada, porém na prática foi possível obter sem muito esforço, devido a forma de implementação da técnica de variabilidade sobre modelo de *features*; (iv) o *information hiding* na literatura para componentes e serviços oculta detalhes da implementação através de uma interface, o que não acontece ao utilizar os comandos do CLI do *framework Ionic*; e, (v) a uniformidade para componentes e serviços tornou-se parcial, tendo em vista que na literatura é caracterizada como total. Essas discordâncias podem ser justificadas por três fatores: diferente complexidade de projetos teóricos e práticos; erros de implementação das técnicas; e limitações da tecnologia utilizada na aplicação das técnicas de implementação

de variabilidade.

Observam-se que as técnicas de implementação de variabilidade definidas e explicadas na literatura, nem sempre possuem o mesmo comportamento quando aplicado a um projeto de contexto real. Quando se trata de um projeto em ambiente controlado, como por exemplo o projeto piloto, o desenvolvedor possui a liberdade para alterar requisitos e escopo do projeto de forma a adequar às necessidades das técnicas.

Em um projeto de estudo de caso, sistema de configuração de roteadores, podem existir requisitos e escopo que não necessariamente são definidas pelo desenvolvedor, mas pelo cliente, acarretando em um aumento na complexidade. Dessa forma, adaptações podem ser necessárias para a implementação das técnicas de variabilidade, como por exemplo, a implementação das técnicas de padrões de projeto no projeto de estudo de caso. Não foi possível implementar as técnicas de variabilidade sobre as *features* e sim, sobre a forma de montagem das *features*. No final, não deixa de ser uma implementação de variabilidade em LPS, mas de fato, não segue a orientação de Apel *et al.* (2013).

A técnica de implementação de variabilidade parâmetros torna-se viável se a equipe de desenvolvimento desconhece o conceito de padrões de projetos e deseja uma implementação variável sem muito esforço de planejamento. Na seção 4.2.1 viu-se que é possível obter um projeto organizado e modular utilizando apenas estruturas condicionais. Em contraste, a utilização de padrões de projeto sobre as técnicas padrão *template-method* e *strategy* atinge uma maior quantidade de critérios de avaliação mais fácil em comparação com a técnica parâmetros. No entanto, tanto o esforço de pré-planejamento quanto de implementação acabam encarecendo a técnica.

Percebe-se que a técnica componentes e serviços para os *frameworks Angular* e *Ionic*, podem ser combinadas com outras técnicas de implementação de variabilidade. Nesse contexto, quando há um conflito sobre critério de avaliação, deve-se especificar em qual perspectiva a técnica será avaliada, ou componentes e serviços ou a técnica combinada.

Dessa forma, componentes e serviços associados a qualquer uma das técnicas de implementação de variabilidade de padrões de projeto é uma candidata apropriada para o desenvolvimento do projeto do estudo de caso. Como o projeto possui uma quantidade relativamente grande de requisitos, em comparação com o projeto piloto, a utilização de componentes e serviços tornam o sistema modular ao nível de pacotes e o padrão *template-method* ou *strategy* tornam o sistema modular ao nível de classes facilitando a depuração, detecção de erros, reutilização de código e o desenvolvimento em LPS.

5

Conclusão

Inicialmente identificaram-se as técnicas de implementação de variabilidade em LPS através de uma revisão da literatura e desenvolveu-se um projeto piloto analisando sua aplicabilidade nos *frameworks* JS, *Angular* e *Ionic*. Posteriormente avaliou-se o uso das técnicas, em um projeto de contexto real, através de um estudo de caso e critérios de avaliação com um protocolo de estudo de caso definido.

Sendo componentes e serviços associado à técnica de implementação de variabilidade *template-method* ou *strategy* uma candidata apropriada para o desenvolvimento do projeto do estudo de caso, por conseguir atender uma maior quantidade de critérios de avaliação. No entanto, componentes e serviços podem ser combinadas com a técnica parâmetros se o foco for atender a baixo planejamento prévio e tempo de desenvolvimento.

5.1 Principais Contribuições

Como não houveram evidências na literatura que reportassem o uso das técnicas de implementação de variabilidade nos *frameworks Angular* e *Ionic*, este trabalho torna-se relevante para a comunidade acadêmica, tendo em vista que a análise realizada das técnicas foi sobre um projeto de contexto real, permitindo assim, a validação das características de cada técnica de implementação de variabilidade encontrada na literatura.

Por outro lado tem-se também como contribuição deste trabalho, a descrição detalhada do protocolo do estudo de caso, possibilitando que pesquisadores possam replicar o estudo em outros contextos ou cenários.

5.2 Trabalhos Futuros

Como trabalhos futuros, têm-se:

- Validação da técnica de implementação de variabilidade candidata ao desenvolvimento do projeto em LPS com profissionais da empresa através de entrevistas;

- Aplicação das técnicas de implementação de variabilidade com outros *frameworks* JS, como o *React* e *Vue.js*;
- Avaliação das técnicas de implementação de variabilidade em outros projetos de contexto real.

Referências

ALVIN, L. F. M. **An Assessment On Variability Implementation Techniques In Software Product Lines: A Replicated Case Study**. Dissertação (Mestrado) — Universidade Federal da Bahia, Salvador, BA, 8 2016.

APEL, S. *et al.* **Feature-Oriented Software Product Lines**. 1. ed. New York: Springer, 2013.

BOSCH, J. *et al.* Variability issues in software product lines. In: SPRINGER. **International Workshop on Software Product-Family Engineering**. [S.l.], 2001. p. 13–21.

CHENG, B. *et al.* Software engineering for self-adaptive systems: A research road map. **Software Engineering for Self-Adaptive Systems, Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee (Eds.). Lecture Notes In Computer Science**, v. 5525, 2009.

DIJKSTRA, E. W. On the role of scientific thought. In: **Selected writings on computing: a personal perspective**. [S.l.]: Springer, 1982. p. 60–66.

FAIN, Y.; MOISEEV, A. **Angular 2 Development with TypeScript**. [S.l.]: Manning Publications Co., 2016.

FLANAGAN, D. **JavaScript: the definitive guide**. [S.l.]: "O'Reilly Media, Inc.", 2006.

GRIFFITH, C. **Mobile App Development with Ionic, Revised Edition: Cross-Platform Apps with Ionic, Angular, and Cordova**. [S.l.]: "O'Reilly Media, Inc.", 2017.

GUTIERREZ, R. M. V.; ALEXANDRE, P. V. M. Complexo eletrônico: introdução ao software. Banco Nacional de Desenvolvimento Econômico e Social, 2004.

IONIC. **Hybrid vs. Native: An introduction to cross-platform hybrid development for architects and app development leaders**. 2018. Disponível em: <https://ionicframework.com/books/hybrid-vs-native>. Acesso em: 11 fev. 2019.

JAIN, N.; BHANSALI, A.; MEHTA, D. Angularjs: A modern mvc framework in javascript. **Journal of Global Research in Computer Science**, v. 5, n. 12, p. 17–23, 2015.

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

MERRIAM, S. B. **Qualitative research: A guide to design and implementation**. [S.l.]: Jossey-Bass higher and adult education series. Jossey-Bass, 2009.

MIKKONEN, T. Formalizing design patterns. In: IEEE. **Proceedings of the 20th international conference on Software engineering**. [S.l.], 1998. p. 115–124.

POHL, K.; BOCKLE, G.; LINDEN, F. J. v. d. **Software Product Line Engineering: Foundations, Principles and Techniques**. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN 3540243720.

PREE, W.; GAMMA, E. **Design patterns for object-oriented software development**. [S.l.]: Addison-wesley Reading, MA, 1995. v. 183.

RASMUSSEN, C. **Hybrid vs Native app development**. 2018. Disponível em: <https://www.nodesagency.com/dk/wp-content/uploads/sites/3/2018/09/Nodes-presentation-The-future-of-app-platforms.pdf>. Acesso em: 25 fev. 2019.

RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. **Empirical software engineering**, Springer, v. 14, n. 2, p. 131, 2009.

SILVA, F. A. P. da *et al.* Linhas de produtos de software: Uma tendência da indústria. 2011.

STACK OVERFLOW. **Developer Survey Results 2017**. 2017. Disponível em: <https://insights.stackoverflow.com/survey/2017>. Acesso em: 23 fev. 2019.

_____. **Developer Survey Results 2018**. 2018. Disponível em: <https://insights.stackoverflow.com/survey/2018/>. Acesso em: 12 nov. 2018.

_____. **Developer Survey Results 2019**. 2019. Disponível em: <https://insights.stackoverflow.com/survey/2019>. Acesso em: 07 jun. 2019.

VALE, T. F. **Using a Multi-Method Approach for Evaluating Service Identification Methods in Service-Oriented Product Lines**. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, BA, 3 2012.

WAKATIME. **Frequently Asked Questions**. 2013. Disponível em: <https://wakatime.com/faq>. Acesso em: 07 jun. 2019.

Anexos

1 Pilot project: sistema de exame online

Esta seção apresenta os detalhes do projeto piloto usados como base para implementação com as estruturas JavaScript. A execução do piloto consiste na implementação de um sistema de exame online. Exame online refere-se à realização de um teste online para medir o conhecimento dos participantes sobre um determinado tópico. Antigamente, todos tinham que se reunir em uma sala de aula ao mesmo tempo para fazer um exame. Com o exame online, os alunos podem fazer o exame online, em seu próprio horário e com seu próprio dispositivo, independentemente de onde estejam. Requer apenas um browser/tablet/smartphone com conexão à Internet.

Nesse sistema de exames, os professores podem criar questões e adicioná-las ao exame. Eles podem escolher entre questões de escolha única/múltipla ou questões de texto livre. Os alunos recebem um link para o exame online, identificando-se para fazer o exame. Eles veem os resultados imediatamente depois.

As funcionalidades refinadas são especificadas como casos de uso. O formato de especificação possui os seguintes elementos: *actor*, *description*, *pré-condition*, *input data*, *main steps*, *alternative steps*, *business rules*, *result/output data*.

- ***Identificador***: representa o ID e o título exclusivos do caso de uso;
- ***Actor***: partes interessadas envolvidas na execução de tal funcionalidade;
- ***Description****:* objetivo(s) principal do requisito;
- ***Pre-condition***: condições a serem cumpridas antes da execução do caso de uso;
- ***Input data***: informações de entrada necessárias para executar a funcionalidade;
- ***Main steps***: fluxo de execução regular do caso de uso;
- ***Alternative steps***: fluxo (s) alternativo (s) que poderiam ser acionados devido a condições ambientais específicas;
- ***Business rules***: restrições de negócios que afetam a execução do caso de uso;
- ***Result/output data***: resultado esperado após a realização dos passos principais;

1.1 Especificações de requisitos

Os atores envolvidos na especificação de casos de uso são explicados na Tabela 1.

Actor	Description
Professor	Responsável por gerenciar a criação e aplicação de questões e exames
Aluno	Responde às questões colocadas pelos professores nos exames

Tabela 1. Use cases actors.

UC001: Criar questão

***Actor*:** professor

Description: o professor cria uma questão e as possíveis respostas a ela.

Pre-condition: N/A

Input data:

Feature obrigatória: texto da questão, categoria da questão, opção(ões) de resposta(s), resposta(s) esperada(s)

***Feature opcional*:** nível da questão

Main steps:

1. O usuário fornece o texto da questão;
2. Informa a categoria;
3. Fornece a(s) opção(ões) de resposta(s);
4. Indica a(s) resposta(s) esperada(s); e
5. Envia a questão.

Alternative steps:

Passo 1. O texto da questão está vazia:

A. O sistema informa ao professor: “Por favor, forneça um texto de questão válido.”

Passo 2. A categoria da questão está vazia:

B. O sistema informa ao professor: “Por favor, forneça uma categoria de resposta válida.”

Passo 3. O texto da resposta está vazio:

C. O sistema informa ao professor: “Por favor, forneça um texto de resposta válida.”

Business rules:

- O texto da questão não pode estar vazio;
- A categoria não pode estar vazia;
- A questão tem apenas uma resposta possível;
- Existe no mínimo 2 e no máximo 10 opções de resposta;
- O sistema deve gerar um ID exclusivo da questão.

Result/output data: questão criada

UC002: Atualizar questão

Actor: professor

Description: o professor atualiza uma questão existente ou as possíveis respostas a ela.

Pre-condition: N/A

Input data:

Feature obrigatória: questão selecionada, texto da questão, categoria da questão, resposta(s) esperada(s)

***Feature opcional*:** nível da questão

Main steps:

1. O usuário escolhe uma questão para editar;
2. O usuário fornece o texto da questão;
3. Informa a categoria;
4. Fornece a(s) opção(ões) de resposta(s);
5. Indica a resposta esperada; e
6. Envia a questão.

Alternative steps:

Step 1. O texto da questão está vazio:

A. O sistema informa ao professor: “Por favor, forneça um texto de questão válida.”

Step 2. A categoria da questão está vazia:

B. O sistema informa ao professor: “Por favor, forneça uma categoria de resposta válida.”

Step 3. O texto da resposta está vazio:

C. O sistema informa ao professor: “Por favor, forneça um texto de resposta válida.”

Business rules:

- O texto da questão não pode estar vazio;
- A categoria não pode estar vazia;
- A questão tem apenas uma resposta possível;
- Existem no mínimo 2 e no máximo 10 opções de resposta.

Result/output data: questão atualizada

UC003: Pesquisar questão

Actor: professor

Description: o professor fornece uma string de pesquisa para encontrar uma ou mais questões que ele está interessado.

Pre-condition: N/A

Input data:

Feature obrigatória: string de pesquisa

Feature opcional**:** N/A

Main steps:

1. O professor fornece uma string de pesquisa;
2. Submete a pesquisa; e
3. O sistema apresenta as questões resultantes.

Alternative steps:

Step 1. A string de pesquisa está vazia:

A. O sistema apresenta todas as questões criadas.

Business rules: o sistema deve pesquisar questões pelo texto da questão, pela categoria da questão e pelas opções de resposta que correspondem a uma ou mais das strings.

Result/output data: lista resultante de questões (apresentando apenas os textos das questões).

UC004: Visualizar detalhes da questão

Actor: professor

Description: quando o professor seleciona uma questão no resultado da pesquisa, ele visualiza os detalhes da questão e a opção de atualizar a questão é ativada.

Pre-condition: professor realizou a pesquisa

Input data:

Feature obrigatória: questão selecionada

***Feature opcional*:** N/A

Main steps:

1. O professor seleciona a questão; e
2. Visualiza os dados.

Alternative steps:

Step 2. O usuário escolhe a opção de atualizar:

A. O sistema redireciona o usuário para a feature Atualizar questão para a questão selecionada

Business rules: N/A

Result/output data: detalhes da questão

UC005: Criar exame

Actor: professor

Description: o professor pesquisará as questões que serão selecionadas para compor o exame. Ele também tem a alternativa de remover a questão selecionada do exame, se necessário.

Pre-condition: questões criadas

Input data:

Feature obrigatória: questões

***Feature opcional*:** ordenar pelo nível da questão

Main steps:

1. O professor informa o nome do exame;
2. Ele pesquisa a questão para ser adicionada;
3. Adiciona a questão;
4. Repete o passo 1 para adicionar mais questões, se necessário;
5. Submete a criação do exame; e
6. O sistema informa a chave de acesso do exame a ser aplicada.

Alternative steps:

Step 2. O professor não fornece o nome do exame:

A. O sistema informa ao professor: “Por favor, forneça um nome do exame válido.”

Step 3. O professor escolhe remover uma questão adicionada:

B. O sistema apresenta uma lista atualizada sem a questão removida.

Step 4. Não há questões selecionadas:

C. A. O sistema informa ao professor: “Você precisa selecionar pelo menos uma questão para criar o exame.”

Business rules: pelo menos uma questão deve ser selecionada e o nome do exame não pode ser vazio.

Result/output data: exame criado

UC006: Resposta do exame

Actor: aluno

Description: o aluno fornece a chave de acesso ao exame ou o seu usuário e senha para acessar a página do exame, responder às questões e visualizar os resultados.

Pre-condition: a chave de acesso do exame é válida.

Input data:

Feature obrigatória* || *alternativa: chave de acesso do exame **||** usuário e senha

***Feature opcional*:** N/A

Main steps:

1. O aluno fornece a chave de acesso do exame;
2. Informa seu nome;
3. Fornece as respostas para as questões;
4. Envia as respostas; e
5. Visualiza o resultado.

Alternative steps:

Step 1. A chave de acesso do exame não existe:

A. O sistema informa ao aluno: “Por favor, forneça uma chave de acesso válida.”

Step 2. O estudante não fornece seu nome:

B. O sistema informa ao aluno: “Por favor, forneça um nome válido.”

Business rules:

- O nome do aluno não pode ser vazio;
- É permitido enviar um exame com respostas vazias;
- O resultado do exame deve informar o percentual e quantidade de questões com respostas certas;
- Também deve apresentar o texto e a(s) resposta(s) selecionada(s) relacionadas aos erros do aluno; e
- O sistema deve registrar a hora de envio da resposta.

Result/output data: resultado do exame