UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

STÉFANY BISKUP COELHO DA SILVA

# Análise do Uso de Boas Práticas no Front-end de um Sistema Web: um Estudo de Caso com Vue.js

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software e Inovação

Orientador: Prof. Dr. Ingrid Oliveira de Nunes

Porto Alegre
2020

**CIP — CATALOGAÇÃO NA PUBLICAÇÃO**

DA SILVA, STÉFANY BISKUP COELHO

Análise do Uso de Boas Práticas no Front-end de um Sistema Web: um Estudo de Caso com Vue.js / STÉFANY BISKUP COELHO DA SILVA. – Porto Alegre: PPGC da UFRGS, 2020.

54 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Orientador: Ingrid Oliveira de Nunes.

1. Qualidade de Software. 2. Manutenibilidade. 3. Boas Práticas. 4. Sistemas Web. 5. Vue.js. I. de Nunes, Ingrid Oliveira. II. Título.

*“Clean code always looks like*
*it was written by someone who cares.”*
— MICHAEL FEATHERS

## RESUMO

Aspectos de qualidade garantem a eficiência de um software desenvolvido. Para tanto, é essencial que os desenvolvedores estejam atentos às especificações e aos padrões prédefinidos, responsabilizando-se pela entrega de um produto que satisfaz todas as expectativas e necessidades dos usuários finais. Ademais, é preciso garantir que este sistema tenha uma boa manutenibilidade, tratando-se de um fator que valida a facilidade na qual aquele sistema está disposto a receber melhorias, adaptações e evoluir, levando em consideração, também, o quanto os componentes desenvolvidos estão coesos e acoplados entre si. O uso de *frameworks* JavaScript e ferramentas tradicionais, como *linters* e tipagem estática, ajudam neste processo de desenvolvimento, porém não previnem todos os problemas. Portanto, o presente trabalho tem como objetivo analisar a qualidade no desenvolvimento *front-end* de um sistema *web* de centralizador de produtos, desenvolvido durante o período de quatro meses, com base nas boas práticas do *framework* Vue.js e ferramentas auxiliares como Vuex, Vue Router e TypeScript, apresentando maneiras de aprimoramento das implementações realizadas e, também, complementar com entrevistas feitas com os especialistas de software envolvidos no desenvolvimento do sistema alvo, com o intuito de compreender o quanto estão familiarizados com as boas práticas do Vue.js.



**Palavras-chave:** Qualidade de Software. Manutenibilidade. Boas Práticas. Sistemas Web. Vue.js.

**Analysis of the Use of Good Practices at the Front-end of a Web System: a Case Study with Vue.js**

## ABSTRACT

Quality aspects guarantee the efficiency of a developed software. Thus it is essential that developers are attentive to the pre-defined specifications and standards, also being responsible for the delivery of a product that satisfies all the expectations and needs of the users. Aside from that, it is necessary to ensure that this system has good maintainability, as it is a factor that validates the ease in which this system is willing to receive improvements, adaptations and evolve, as well as taking into consideration how cohesive the developed components are and how coupled they are to each other. The use of JavaScript frameworks and traditional tools, such as linters and static typing, help in the development process, but do not prevent all the problems. Therefore, the present paper aims to analyze the quality of the front-end development of a product centralizer web system, developed over a period of four months, based on the good practices of framework Vue.js and auxiliary tools such as Vuex, Vue Router and TypeScript, presenting ways to improve the implementations made and, moreover, it complements with interviews made with software specialists involved in the development of the target system, in order to understand how familiar they are with the Vue.js good practices.

**Keywords:** Software Quality, Maintainability, Good Practices, Web Systems, Vue.js.

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| SM | Scrum Master |
| TI | Tecnologia da Informação |
| API | Application Programming Interface |
| CLI | Command-Line Interface |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| DRY | Don’t Repeat Yourself |
| IDE | Integrated Development Environment |
| MVP | Minimum Viable Product |
| POC | Proof of Concept |
| SPA | Single-Page Application |
| CRUD | Create, Read, Update and Delete |
| HTML | HyperText Markup Language |
| MVVM | Model-View-ViewModel |
| REST | Representational State Transfer |
| FURPS | Functionality, Usability, Reliability, Performance, Supportability |

## LISTA DE FIGURAS

## LISTA DE TABELAS

**SUMÁRIO**

# 1 INTRODUÇÃO

Em projetos de desenvolvimento de sistemas, existem alguns aspectos que devem ser levados em consideração. Um dos mais importantes é a qualidade do software que será construído, em virtude de referir-se ao fator principal de garantia de que todas as funcionalidades estão de acordo com as necessidades dos clientes (BASS; CLEMENTS; KAZMAN, 2012). Isto posto, é fundamental ser capaz de avaliar a sua capacidade de manutenção, tratando-se de um atributo que desempenha um papel significativo no nível de qualidade durante todo o seu ciclo de vida (AGARWAL; TAYAL, 2009).

A manutenibilidade é o nível de aptidão de um sistema em receber modificações, com o intuito de evoluir em menor tempo e custo. Seu principal objetivo é alcançar um código limpo, claro e simples, de fácil compreensão, utilizando técnicas e princípios para desenvolvimento de código (MARTIN, 2008), reduzindo custos e tempo em sua evolução, garantindo uma maior qualidade nas soluções propostas.

É preciso garantir que o sistema satisfaça em objetivos, custo e desempenho de modo funcional. Para esse fim, durante todo o processo de ciclo de vida do software, recomenda-se seguir padrões que atendam às boas práticas e que cumpram todas as etapas necessárias (MARTIN, 2008; AGARWAL; TAYAL, 2009). Dessa forma, é assegurada a qualidade do produto ou serviço entregue aos usuários finais, tornando-o apto para adquirir novas funcionalidades sem arcar com grandes dificuldades.

No entanto, ainda é um desafio entregar projetos com um bom nível de qualidade e manutenibilidade. É um processo que há riscos de atrasos e surgimento de *bugs*, em razão de variabilidades e incertezas que podem vir a surgir. Essas situações podem ocorrer, por exemplo, em circunstâncias nas quais novas tecnologias são empregadas ou experimentadas pela primeira vez pelos desenvolvedores, com intenção de explorar potenciais benefícios (YOURDON, 2004). Ademais, um outro possível contexto é quando, em busca de sucesso e menor concorrência, as empresas procuram maneiras de agir depressa para entregar suas demandas, recorrendo a atalhos de modo a atender objetivos a curto prazo, gerando débitos técnicos (KRUCHTEN; NORD; OZKAYA, 2019), soluções menos elegantes e com critérios mais leves com relação à qualidade, impactando-a e comprometendo-a.

Consequentemente, é produzido um sistema legado (MARTIN, 2008), de pouca qualidade e não atendendo a todos os requisitos, sem uma base sólida para funcionar adequadamente. Desta forma, se sucedem algumas consequências, que influenciam na

performance e integração do software, trazendo complexidades em futuras implementações para evoluí-lo.

Neste estudo, é analisada a qualidade do código fonte no desenvolvimento de um projeto de pequena escala, averiguando o descumprimento de melhores práticas das tecnologias empregadas e apresentando possíveis aperfeiçoamentos. O sistema foi desenvolvido com o *framework* Vue.js[1] e tipagem estática TypeScript[2], além do uso de algumas bibliotecas e ferramentas adicionais.

Para a compreensão adequada desta monografia, os próximos capítulos foram organizadas como segue. O Capítulo 2 traz temas como qualidade de software e conceitos de manutenibilidade, complementando com alguns trabalhos relacionados que avaliam a qualidade de código JavaScript, bem como informações a respeito do uso do *framework* Vue.js. No Capítulo 3 são descritas as justificativas e objetivos da presente pesquisa, que conta com uma análise quantitativa e qualitativa, apresentando os resultados do estudo no Capítulo 4. Como desfecho, no Capítulo 5, apresentam-se as conclusões obtidas.

[1] Vue.js — Disponível em: <https://vuejs.org>

[2] TypeScript: Typed JavaScript at Any Scale — Disponível em: <https://www.typescriptlang.org>

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo contempla explicações a respeito de aspectos de qualidade de software, como manutenibilidade e suas características, além de informações acerca das tecnologias empregadas para o desenvolvimento da aplicação em questão. Ademais, são mostrados casos nos quais alguns autores analisam sistemas que utilizam JavaScript como linguagem de programação.

## 2.1 Qualidade e Manutenibilidade de Software

A área de engenharia de software possui diversos propósitos, e um deles é prover boas práticas em busca de melhorar a qualidade dos produtos desenvolvidos, tanto em seu processo quanto em sua entrega aos usuários finais. No sentido de certificar a aptidão de um sistema, é preciso atender a algumas premissas e especificações. Conforme conceitua a norma internacional da ISO/IEC JTC 1/SC 7 (2017), qualidade de software pode ser entendida, em tradução livre, como o "nível em que um produto de software atende aos requisitos previamente estabelecidos".

Portanto, no quesito de software, existe a divisão atendendo a alguns fatores acerca de qualidade: internas e externas (AGARWAL; TAYAL, 2009). No que se diz respeito aos fatores de qualidade internos, são representados aqueles sobre o cumprimento de boas práticas e propriedades estruturais de um sistema; detalhes normalmente notados pelos desenvolvedores. Em contrapartida, os fatores de qualidade externos são evidenciados pelo ponto de vista do usuário, que devem cumprir com necessidades e especificações previamente solicitadas, além de ser necessário ter um bom nível na funcionalidade, usabilidade, confiabilidade, desempenho e suportabilidade, atributos representados pelo acrônimo FURPS sugerido por Grady e Caswell (1987). Dessa forma, deve-se atentar para ambas classificações, visando assegurar que o produto satisfaz todas as expectativas.

Por fim, existem diferentes categorias com relação às suas características, sendo estas avaliadas por meio de métricas específicas, que correspondem a: adequação funcional, eficiência de desempenho, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade (ISO/IEC JTC 1/SC 7, 2011). Neste trabalho, foca-se em uma característica específica de qualidade de software, chamada *manutenibilidade*, que é definida como, em tradução livre, a "facilidade com que um sistema ou componente pode ser modificado para corrigir falhas, melhorar o desempenho ou se adaptar a um am-

Figura 2.1 – Subcaracterísticas da Manutenibilidade

| | Descrição |
|---|---|
| Modularidade | Grau no qual um sistema ou programa de computador é composto de componentes discretos, de forma que uma alteração em um componente tenha impacto mínimo em outros componentes. |
| Modificabilidade | Grau no qual um produto ou sistema pode ser modificado de forma eficaz e eficiente sem introduzir defeitos ou degradar a qualidade do produto existente. |
| Reutilização | Grau em que um ativo pode ser usado em mais de um sistema ou na construção de outros ativos. |
| Analisabilidade | Grau de eficácia e eficiência com o qual é possível avaliar o impacto em um produto ou sistema de uma alteração pretendida em uma ou mais de suas peças, ou diagnosticar um produto quanto a deficiências ou causas de falhas, ou identificar peças para ser modificado. |
| Testabilidade | Grau de eficácia e eficiência com o qual os critérios de teste podem ser estabelecidos para um sistema, produto ou componente e os testes podem ser realizados para determinar se esses critérios foram atendidos. |

Fonte: Adaptado de ISO/IEC JTC 1/SC 7 (2011)

biente" (IEEE, 1990). A manutenibilidade possui uma série de subcaracterísticas, que descrevem fatores internos de qualidade de software, como é possível observar na Figura 2.1.

Considerando um sistema típico, que evolui continuamente para atender às frequentes e progressivas necessidades de seus usuários, as tarefas de manutenção são imprescindíveis durante o seu ciclo de vida. De acordo com as duas primeiras leis criadas por Lehman (1980), "mudança contínua"e "complexidade crescente", a evolução de um sistema está atrelada à ideia de precisar ser continuamente adaptado, para que não se torne insatisfatório e continue útil e relevante. Tais adaptações incluem correções de erros, aumento de desempenho e outras melhorias, como novas funcionalidades e inserção de testes (AGARWAL; TAYAL, 2009). No entanto, conforme as adaptações de melhorias são realizadas, o sistema se torna mais complexo, necessitando de um esforço extra para que não gerar complicações futuras.

Algumas maneiras de melhorar e facilitar a manutenção de um sistema têm relação com a componentização e a reutilização (SAMETINGER, 2013): decompor regras de negócio em pequenas funções para cumprirem seu papel, livre de ambiguidades (MARTIN, 2008), consistindo em simplificar projetos separando suas partes lógicas de código para serem reutilizáveis de maneira planejada e sistemática. Assim, ocorre uma redução de esforços e o princípio *Don't Repeat Yourself* (DRY) é seguido: cada pedaço de conhecimento com uma única e confiável representação dentro de um sistema (HUNT; THOMAS, 1999).

Nesse âmbito, algumas definições de fatores que afetam diretamente a manutenção surgiram, como acoplamento e coesão (AGARWAL; TAYAL, 2009), que são usados como avaliadores da qualidade interna e confiabilidade do software.

As métricas de acoplamento correspondem ao grau de interação e associação estabelecidos por meio da ligação entre seus componentes — o quanto estão conectados e são dependentes para funcionar. Dessa forma, um alto nível de acoplamento quer dizer um aumento na complexidade, por conta de uma maior dependência.

O conceito de coesão compreende o grau em que um componente está focado em realizar apenas uma única tarefa. São levados em consideração os conceitos de *single-responsibility* dos princípios SOLID criados por Martin (2008) e a ideia de manter a independência por meio da escrita de "código tímido" (HUNT; THOMAS, 1999). Ou seja, assumir responsabilidades que não são próprias do componente é um cenário insatisfatório. Portanto, uma baixa coesão significa que o componente realiza muitas tarefas que não deveria.

Uma vez ignorados esses fatores, alguns problemas surgem, como dificuldades de gerenciamento, manutenção e reuso (AGARWAL; TAYAL, 2009). Além desses problemas, um alto acoplamento e uma baixa coesão levam a grandes chances de instabilidade no sistema, visto que os componentes se tornam muito dependentes entre si e, portanto, são constantemente afetados por quaisquer mudanças. Já aqueles com baixo acoplamento e alta coesão são mais fáceis de serem mantidos e, consequentemente, geram menor custo para implementação de funcionalidades ou correção de erros.

## 2.2 Estudos sobre Qualidade de Código em JavaScript

Alguns autores vêm publicando estudos a respeito de problemáticas em código JavaScript há alguns anos. Por exemplo, Ocariza Jr., Pattabiraman e Zorn (2011) focam principalmente em identificar erros comuns em aplicações *web* e suas características. De maneira mais aprofundada que neste trabalho, Ocariza et al. (2013) e Ocariza et al. (2017) procuram analisar projetos com a intenção de gerar relatórios de *bugs* para, então, examiná-los e identificar as causas dessas falhas, assim como suas consequências.

Outros procuram encontrar falhas por meio de ferramentas, como Saboury et al. (2017). Eles descrevem um estudo de caso que aplica a ferramenta SmellJS para analisar o código fonte em busca de alguns tipos de *code smells*, em diversas versões de cinco aplicações populares desenvolvidas com JavaScript.

Em suma, a grande maioria dos equívocos identificados em código fonte de sistemas *web* foram causados por erros introduzidos manualmente pelos programadores (Ocariza et al., 2013). Por outro lado, os erros detectados são aqueles que, por vezes, são

gerados por falta de uso de ferramentas auxiliares que identificam previamente certos descuidos, como foi mostrado ao implementar o SmellJS (Saboury et al., 2017).

Neste projeto, o uso de uma ferramenta *linter*, que identifica e relata falta de padrões em JavaScript quando instalada em editores de código, e o uso de tipagem estática TypeScript, que fornece tipos a trechos de código e valida se está tudo semanticamente correto, já previnem essas questões desde o princípio. Na prática, os desenvolvedores envolvidos no desenvolvimento do sistema alvo precisam lidar com outros problemas que *linters* e tipagem estática não conseguem detectar: aqueles relacionados às melhores práticas quanto ao uso de *frameworks*, como o Vue.js, e suas bibliotecas auxiliares.

## 2.3 *Framework* Vue.js

No atual contexto tecnológico, diversas novas aplicações estão gradualmente se tornando mais modernas e poderosas como plataformas *web*, sendo a utilização de apenas JavaScript puro, sem bibliotecas auxiliares e *frameworks*, insuficiente para resolver de modo facilitado e eficiente tudo que é preciso e desejado.

O Vue.js[1] é um *framework* JavaScript reconhecido, e que está amadurecendo aceleradamente com relação à sua popularidade e sua aprovação diante a comunidade *open source* de programação (MACRAE, 2018). Neste projeto, foi optado pelo seu uso pois possui (WOHLGETHAN, 2018):

- Estrutura baseada em componentes, sendo relevante para facilitar a manutenibilidade do sistema;
- Práticas aceitas, oferecendo-as em sua documentação para uma configuração de desenvolvimento completa;
- Facilidade de integração com outras tecnologias e bibliotecas de maneira progressiva; e
- Pequena curva de aprendizado.

Ele foi disponibilizado no início do ano 2014 por Evan You, antigo programador da Google, tornando-se uma proposta *open source* bem recebida no mercado de trabalho. Seu objetivo foi apresentar uma proposta que houvesse os melhores aspectos de outros dois famosos *frameworks* no mercado, denominados AngularJS[2] e React[3].

[1] Vue.js — Disponível em: <https://vuejs.org>
[2] AngularJS — Superheroic JavaScript MVW Framework — Disponível em: <https://angularjs.org>
[3] React – A JavaScript library for building user interfaces — Disponível em: <https://reactjs.org>

A proposta é baseada no padrão de arquitetura *Model-View-ViewModel* (MVVM) (HANCHETT; LISTWON, 2018). Ele compreende regras de negócio e dados indispensáveis na camada *Model*, e os apresenta em interfaces na camada de *View* por meio da conexão que é realizada na camada *ViewModel*. O padrão leva em consideração, ainda, conceitos de reatividade e componentização (HANCHETT; LISTWON, 2018).

Para iniciar a criação de novas aplicações *web* com o Vue.js, é sugerido o uso da ferramenta auxiliar Vue CLI[4]. Essa ferramenta de interface de linha de comando, mediante a uma série de sugestões de recursos disponíveis, inicializa um projeto em questão de poucos minutos, padronizando o ambiente de trabalho e, além disso, integrando o sistema com outras ferramentas úteis para o desenvolvimento.

Para mais informações sobre Vue.js, o *framework* possui uma vasta e completa documentação que pode ser acessada em (YOU, 2017).

## 2.4 Considerações Finais

Em resumo, diversos conceitos a respeito de engenharia de software e melhores práticas precisam ser levados em consideração durante todo o processo de desenvolvimento de sistemas, com o intuito de não prejudicar a qualidade e a manutenibilidade do que é construído. Muitas vezes, *frameworks open source* disponíveis no mercado ajudam a tornar esse processo mais eficiente.

Porém, nem sempre é possível seguir padrões, sendo alguns difíceis de detectar suas violações por meio de métricas tradicionais, o que acaba tornando o código difícil de manter, adaptar e melhorar. Nesse sentido, é preciso agir de forma a detectar esses possíveis descumprimentos às boas práticas e, dessa forma, corrigi-los para entregar produtos e serviços com mais qualidade aos usuários finais.

[4]Vue CLI — Disponível em: <https://cli.vuejs.org>

# 3 DETALHAMENTO DE ESTUDO

O objetivo do presente trabalho baseia-se em avaliar as violações às boas práticas quanto ao uso das tecnologias adotadas e ouvir a opinião dos especialistas de software envolvidos no desenvolvimento do sistema alvo. Assim, é apresentada a maneira ideal de desenvolvimento com possíveis formas de aperfeiçoamentos e os relatos dos entrevistados. Desse modo, a pergunta referente ao problema consiste em: como é o uso de boas práticas na implementação do *front-end* em Vue.js em um projeto comercial?

## 3.1 Sistema Alvo

Os dados neste estudo vigente foram obtidos a partir de um sistema *web* de catálogo centralizado de produtos, que engloba 14 informações a respeito destes, sendo desenvolvido por um time de quatro desenvolvedores de uma empresa internacional, de grande porte, com atuação no mercado há mais de 70 anos. A Tabela 3.1 apresenta dados gerais a respeito do *front-end* do projeto. Em virtude de um acordo de confidencialidade, não é possível divulgar particularidades do sistema além das aqui descritas.

### 3.1.1 Contexto

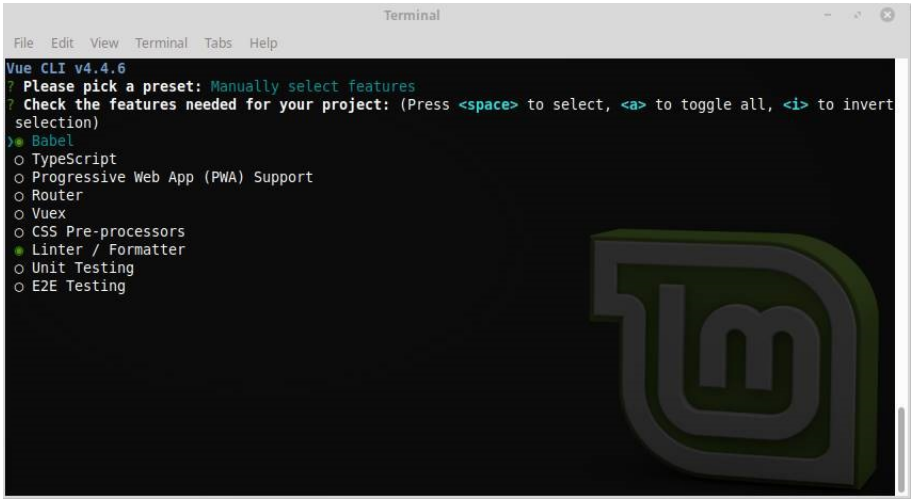
No início do ano de 2020, surgiu um notável interesse na criação de uma *single-page application* (SPA) por parte dos *stakeholders*, que fosse um catálogo centralizado de produtos e que facilitasse, resumidamente, as funcionalidades básicas de *create, read, update* e *delete* (CRUD) de informações a respeito destes. À vista disso, o time fez uma proposta e, após alguns acordos de ambas partes, definiu-se a entrega final para o segundo semestre desse mesmo ano.

Tabela 3.1 – Informações Macro Acerca do Sistema Alvo

| **Informação** | **Total** |
|---|---|
| Número de Arquivos | 62 |
| Número de Funções | 57 |
| Número de Componentes | 7 |
| Número de Propriedades | 9 |
| Linhas de Código | 3698 |

Fonte: Autora

Figura 3.1 – Etapa de Criação de Projetos Por Meio do Vue CLI




Fonte: Autora

Pela disponibilidade de outro especialista da equipe, que trabalha com *back-end*, as *application programming interfaces* (APIs), que o *front-end* realiza chamadas, foram desenvolvidas sem adversidades, não havendo necessidade de avaliar essas melhoras práticas no presente trabalho. Para a implementação destas APIs, são empregadas GraphQL[1] com Python[2] e Django REST *framework*[3], integrando com o banco de dados PostgreSQL[4].

As ferramentas utilizadas para o desenvolvimento do *front-end* do sistema deste presente trabalho, conforme mostrados como sugestões no processo de criação de um projeto, por meio da ferramenta Vue CLI na Figura 3.1, são listadas a seguir.

- Transpilação de código JavaScript para versões adequadas com Babel[5], que converte o código fonte desenvolvido para versões compatíveis de JavaScript, garantindo que os navegadores possam executar os sistemas *web* apropriadamente.
- Determinação do uso de tipagem estática aplicando TypeScript[6], que fornece tipagem em ambiente de desenvolvimento às variáveis, métodos e classes de um sistema. Ademais, gera avisos ao programador durante o processo de desenvolvimento, visto que são verificados os possíveis dados de entrada e saída dos códigos criados instantaneamente.

[1]GraphQL | A query language for your API — Disponível em: <https://www.graphql.org>
[2]Python — Disponível em: <https://www.python.org>
[3]Django REST framework — Disponível em: <https://www.django-rest-framework.org>
[4]PostgreSQL — Disponível em: <https://www.postgresql.org>
[5]Babel · The compiler for next generation JavaScript — Disponível em: <https://babeljs.io>
[6]TypeScript: Typed JavaScript at Any Scale — Disponível em: <https://www.typescriptlang.org>

- Navegação por rotas e acesso aos *navigation guards* por meio do Vue Router[7], que permite navegação por sistemas *web* SPA sem que a página seja atualizada à medida que o usuário navega, e possibilita desencadear funções pertinentes durante diversos pontos da execução e processamento das páginas.
- Padronização de gerenciamento de estado por ações do Vuex[8], que centraliza o estado das aplicações, de modo a se tornar a entidade global única e responsável por comunicar e compartilhar dados a todos seus os componentes, seguindo regras que garantem que essas informações apenas podem ser modificadas de maneira previsível.
- Conversão de CSS com o pré-processador denominado SASS[9], que facilita a dinâmica da escrita de estilos de páginas *web*, fornecendo a possibilidade de criação de variáveis e arquivos, extensão de regras, aninhamento de classes, entre outros;
- Detecção de erros, *warnings* e *bugs* utilizando o *linter* ESLint[10], que automatiza a checagem de código, prevenindo a inserção de erros e encontrando pontos de melhoria durante o processo de desenvolvimento de um sistema.
- Validação de funcionalidades com testes unitários usando o *framework* Jest[11], que simplifica a criação e configuração de testes, com *feedbacks* rápidos e relatórios de cobertura de código.

## 3.2 Procedimento do Estudo

Com o intuito de levantar dados a respeito das violações de convenções no sistema alvo, a análise é feita de duas formas: quantitativa e qualitativa. A análise quantitativa reflete a observação quanto às normas, os padrões e as melhores práticas sobre as tecnologias empregadas no sistema, detectando e avaliando a qualidade no seu uso. Em seguida, na análise qualitativa, são apresentadas as entrevistas realizadas com os colaboradores do projeto, com o intuito de entender a familiaridade do time sobre o Vue.js.

[7] Vue Router — Disponível em: <https://router.vuejs.org>
[8] Vuex — Disponível em: <https://vuex.vuejs.org>
[9] Sass: Syntactically Awesome Style Sheets — Disponível em: <https://sass-lang.com>
[10] ESLint - Pluggable JavaScript linter — Disponível em: <https://eslint.org>
[11] Jest · Delightful JavaScript Testing — Disponível em: <https://jestjs.io>

Tabela 3.2 – Critérios de Boas Práticas

| Critério | Resumo |
|---|---|
| Convenções | Uso adequado do estado da aplicação (*getters*, *mutations* & *actions*). |
| Acoplamento | Componentes de acordo com os princípios de arquitetura multicamadas. |
| Padronização | Uso de constantes. |
| *Lifecycle Hooks* | *Data fetching* (recuperação de dados) de maneira otimizada. |
| Arquivos de Declaração | Organização em múltiplos arquivos. |
| Notação de Nomes de Componentes em Arquivos | Devem ser sempre `PascalCase` ou `kebab-case` e multipalavras. |
| Notação de Nomes de Propriedades em Arquivos | Devem ser sempre `camelCase`. |
| Notação de Nomes de Componentes em Templates | Devem ser sempre `kebab-case` e multipalavras. |
| Notação de Nomes de Propriedades em Templates | Devem ser sempre `kebab-case`. |
| Chave de Identificação no `v-for` | Sempre usar o atributo `key` quando usar `v-for`. |
| Não Uso de `v-if` com `v-for` | Evitar verificações desnecessárias. |
| Estilos em Componentes com Escopo | Garantir que o estilo aplica-se somente aos componentes correspondentes. |
| Elementos Multiatributos | Devem abranger várias linhas, com um atributo por linha. |
| Aspas em Valores de Atributos | Valores de atributos HTML não vazios devem sempre estar dentro de aspas. |
| Abreviação de Diretivas | Devem ser sempre usadas ou nunca usadas, mantendo-se um padrão. |

Fonte: Autora

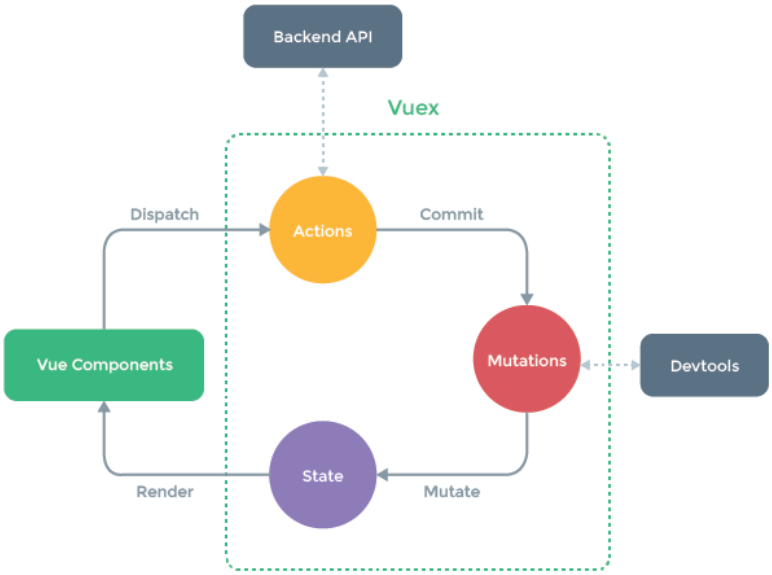
### 3.2.1 Análise Quantitativa

Para estudar a qualidade do código fonte, são escolhidas práticas conhecidas das tecnologias utilizadas e de desenvolvimento de sistemas, como descrito a seguir e resumido na Tabela 3.2. Elas são indicadas pelas documentações do Vue.js[12], do Vue.js *Style Guide*[13], do Vuex[14] e do Vue Router[15], e foram selecionadas de acordo com as particularidades do software sem expor dados sensíveis e confidenciais. A partir desses critérios de boas práticas, é possível analisar manualmente o código fonte do sistema alvo em busca de possíveis descumprimentos.

- **Convenções:** Ao empregar o Vuex, seguindo um padrão de gerenciamento de estado compartilhado, é centralizado o armazenamento de dados do sistema, como indica a Figura 3.2, fazendo com que um ou mais componentes possam acessar um estado único, de acordo com o padrão de design *singleton*. Em razão disto, todos os estados da aplicação estão contidos em um grande objeto (chamado *store*). Essa implementação garante que o estado da aplicação possa ser sempre modificado de forma previsível, assegurando questões de reatividade. Além disso, evita duplicação de código ao apresentar *getters*, *mutations* e *actions*. É preciso, então, empregar Vuex de maneira adequada, implementando-os:

[12]Guide - Vue.js — Disponível em: <https://vuejs.org/v2/guide/>
[13]Style Guide - Vue.js — Disponível em: <https://vuejs.org/v2/style-guide/>
[14]Getting Started | Vuex — Disponível em: <https://vuex.vuejs.org/guide/>
[15]Getting Started | Vue Router — Disponível em: <https://router.vuejs.org/guide/>

Figura 3.2 – Diagrama de Fluxo de Dados com Vuex




Fonte: Vuex *Guide*[16]

- ***Getters*:** São acessados para obter o estado da aplicação e retornar todos os dados desejados;
- ***Mutations*:** São usados para manipular e alterar o estado da aplicação de forma síncrona; e
- ***Actions*:** São empregados para realizar ações de maneira assíncrona antes de efetuar mutações.

- **Acoplamento:** Seguindo a proposta do fluxo adequado de dados do Vuex empregado na aplicação, o objetivo é acessar e recuperar dados por meio da *store*, e não diretamente dos *endpoints* de API, desacoplando os componentes da camada de serviço.
- **Padronização:** Formalizar o uso de constantes em um único arquivo permite que os colaboradores tenham uma visão rápida e centralizada do que está à disposição em todo o projeto, em arquivos como `getter-types.ts`, `mutation-types.ts` e `action-types.ts` de acordo com o que é sugerido na documentação do Vuex e na Figura 3.3.

[16] What is Vuex? | Vuex — Disponível em: <https://vuex.vuejs.org>

Figura 3.3 – Exemplo de Uso de Constantes para Fluxo de Dados Vuex

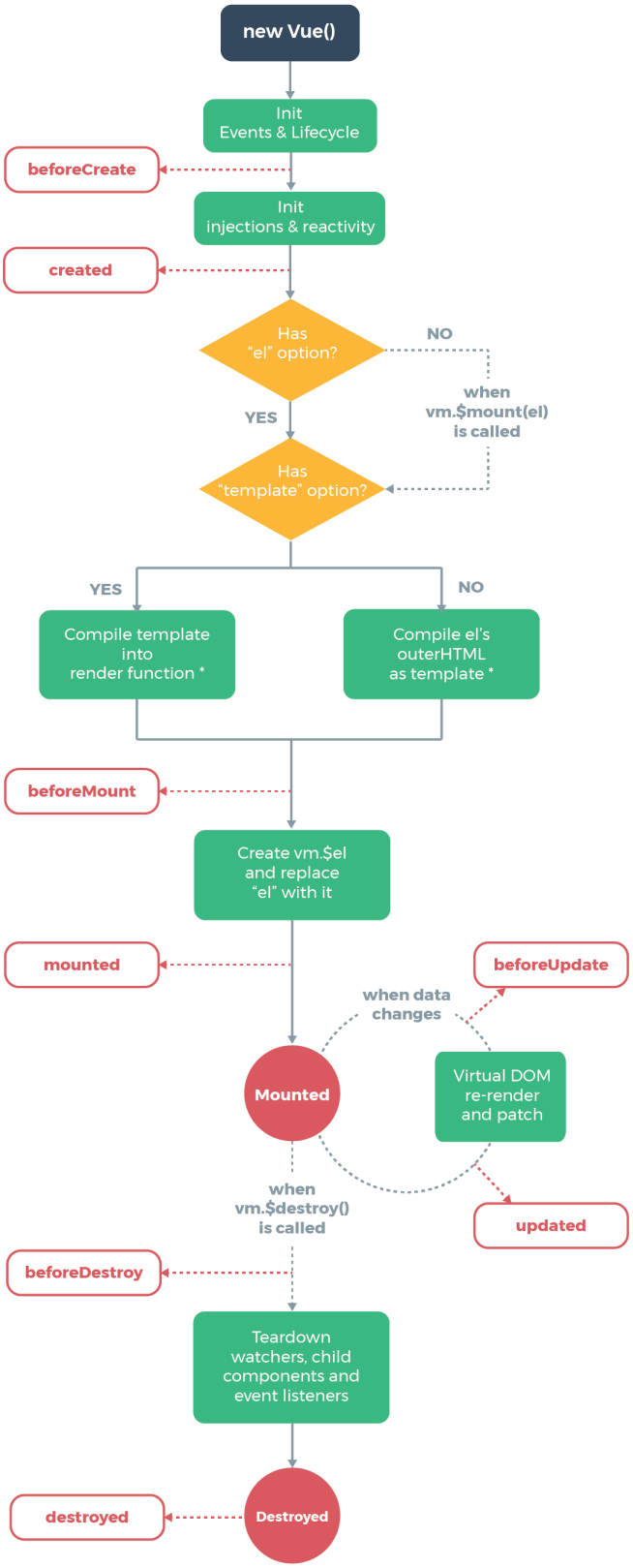
```
// @/store/getter-types.ts
export const GETTER_EXAMPLE = 'getterExample';
// @/store/mutation-types.ts
export const MUTATION_EXAMPLE = 'mutationExample';
// @/store/action-types.ts
export const ACTION_EXAMPLE = 'actionExample';

// @/store/index.ts
import Vuex from 'vuex';
import product from './modules/products';
import { GETTER_EXAMPLE } from './getter-types';
import { MUTATION_EXAMPLE } from './mutation-types';
import { ACTION_EXAMPLE } from './action-types';

const store = new Vuex.Store({
  state: { ... },
  getters: {
    [GETTER_EXAMPLE]: (state) => { // getter state },
  },
  mutations: {
    [MUTATION_EXAMPLE]: (state) => { // mutate state },
  },
  actions: {
    [ACTION_EXAMPLE]: async ({ commit }) => { // action state },
  },
  modules: {
    products,
  },
});
```

Fonte: Adaptado de Vuex Guide[17]

- ***Lifecycle Hooks*:** Há uma sequência de eventos que percorrem diversos pontos ao longo do ciclo de vida de um componente, indicados pela Figura 3.4. Cada ponto pode desencadear chamadas de funções durante sua execução, caso necessário. A maneira otimizada de realizar *data fetching* é com o apoio dos padrões de Vue Router[18]. Com a abordagem *Fetching Before Navigation*, buscamos todos os dados necessários para as respectivas páginas do sistema antes de realmente navegar pela aplicação, o que facilita para lidar com o resultado do processamento, como mensagens de erros, impedimentos e alertas variados.

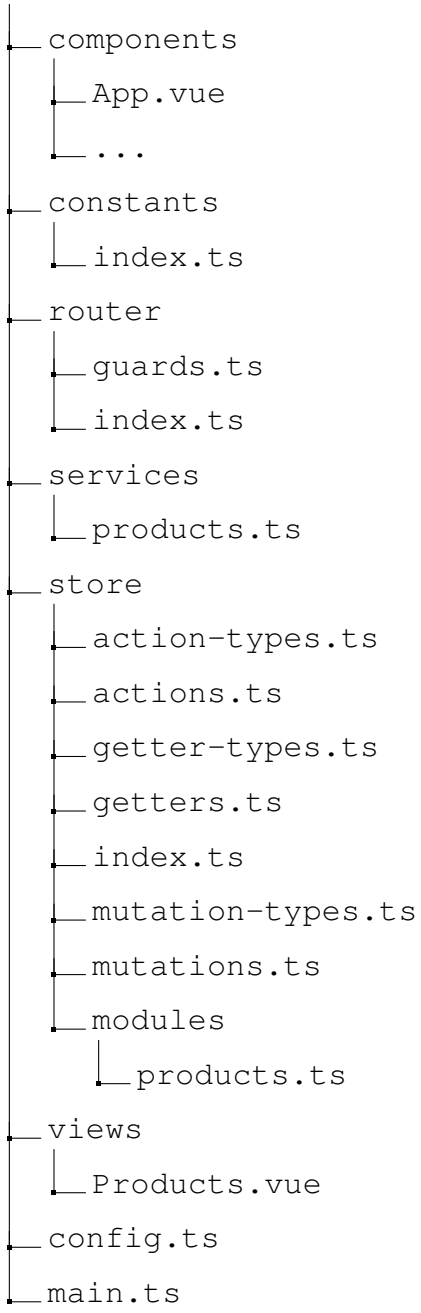
[17]Mutations | Vuex — Disponível em: <https://vuex.vuejs.org/guide/mutations.html>
[18]Data Fetching | Vue Router — Disponível em: <https://router.vuejs.org/guide/advanced/data-fetching.html>
[19]The Vue Instance | Vue.js — Disponível em: <https://vuejs.org/v2/guide/instance.html/>

Figura 3.4 – Diagrama de Ciclo de Vida da Instância Vue.js




Fonte: Vue.js *Guide*[19]

Figura 3.5 – Exemplo de Representação de Árvore da Estrutura da Aplicação

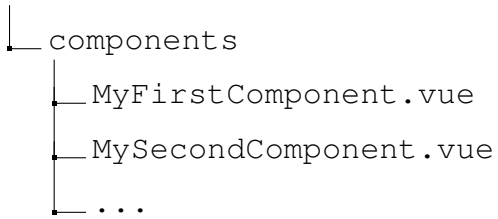



Fonte: Adaptado de Vuex *Guide*[20]

- **Arquivos de Declaração:** Conforme o arquivo de armazenamento vai se desenvolvendo, ele pode acabar se tornando muito extenso. Dessa forma, é possível simplesmente dividir as *actions*, *mutations* e *getters* em arquivos separados, como é sugerida na estrutura de projeto representada pela Figura 3.5.

[20] Application Structure | Vuex — Disponível em: <https://vuex.vuejs.org/guide/structure.html>

Figura 3.6 – Exemplo de Uso de Convenções na Nomeação de Componentes




Fonte: Adaptado de Vue.js *Style Guide*

Figura 3.7 – Exemplo de Uso de Convenção na Nomeação de Propriedades

```
import { Prop } from 'vue-property-decorator';

@Prop({ required: true }) readonly objectsList: object[];
```

Fonte: Adaptado de Vue.js *Style Guide*

Figura 3.8 – Exemplo de Uso de `kebab-case` para Componentes e Propriedades em Templates

```
<template lang="pug">
  my-first-component(:objects-list="[]")
</template>
```

Fonte: Adaptado de Vue.js *Style Guide*

- **Notação de Nomes de Componentes em Arquivos:** Os nomes dados aos componentes devem ser sempre multipalavras, exceto pelo componente raiz `App.vue` e componentes internos fornecidos pelo próprio *framework* Vue. Isto previne conflitos com elementos HTML existentes e futuros, visto que estes são formados por apenas uma única palavra. Além disso, a notação com `PascalCase` garante melhor autocompletação nos editores de código (IDEs), pois é consistente com a forma como os componentes são referenciados. Portanto, é a maneira definida pelos desenvolvedores do projeto, ao invés de `kebab-case`. A união destas duas convenções é vista na Figura 3.6.
- **Notação de Nomes de Propriedades em Arquivos:** A notação com `camelCase` é usada por padrão pelo JavaScript, portanto é previsto seguir a convenção desta linguagem para os nomes das propriedades ao usar Vue, como mostra a Figura 3.7.
- **Notação de Nomes de Componentes e Propriedades em Templates:** Como o HTML é insensível a maiúsculas e minúsculas, os componentes e suas propriedades nos templates *document object model* (DOM) devem usar a notação `kebab-case`, como apresenta a Figura 3.8.

Figura 3.9 – Exemplo de Uso de `v-for` Aplicando Estilo com Escopo

```
<template lang="pug">
  ul.unordered-list(v-if="hasExamples")
    li(
      v-for="example in examples"
      :key="example.id"
    )
      {{ example.description }}
</template>

<style lang="scss" scoped>
  .unordered-list {
    list-style-type: circle;
  }
</style>
```

Fonte: Adaptado de Vue.js *Style Guide*

- **Chave de Identificação e Não Uso de `v-if` com `v-for`:** O atributo `key` (também conhecido como chave de identificação) em `v-for` sempre deve ser usado nos componentes. Isso faz com que o estado interno do componente se mantenha em ordem, com um comportamento previsível. Ademais, o uso de `v-if` para filtrar itens de uma lista ou, ainda, escondê-la, gera malefícios em sua eficiência, já que precisa verificar todos os itens da lista a todo o momento, uma lógica desnecessária para o template: o ideal é realizar verificações antes de iniciar qualquer iteração. Um exemplo do seu bom uso é demonstrado pela Figura 3.9.
- **Estilos em Componentes com Escopo:** Aplicar `scoped` ao CSS tem como objetivo de tornar os estilos internos mais fáceis, podendo conter nomes de classes legíveis e sem precisar de muita especificidade, já que se tornam improváveis de resultar algum conflito com outros componentes: garantem que o que é aplicado é apenas usado pelo componente em questão, como pode ser visto na Figura 3.9.
- **Elementos Multiatributos, Aspas em Valores de Atributos e Abreviação de Diretivas:** Em um template, dividir os seus elementos que possuem vários atributos em várias linhas, e fazer com que estes atributos estejam com os respectivos valores dentro de aspas, é benéfico por conta da legibilidade facilitada. Outra convenção que também melhora a legibilidade, por fim, são as abreviações de diretivas, que devem ser sempre usadas ou nunca usadas, mantendo-se um padrão. As diretivas podem ser `:` para `v-bind:`, `@` para `v-on:` e `#` para `v-slot`, como é visto na Figura 3.10. A equipe optou pelo uso das abreviações para todo o projeto.

Figura 3.10 – Exemplo de Uso de Diretivas Abreviadas em Multiatributos Usando Aspas

```
<template lang="pug">
  template(#header)
    input(
      type="number"
      :placeholder="Type a number"
      @focus="onFocus"
    )
</template>
```

Fonte: Adaptado de Vue.js *Style Guide*

### 3.2.2 Análise Qualitativa

Para complementar o presente trabalho, são realizadas entrevistas a fim de coletar opiniões sobre o conhecimento em desenvolvimento de sistemas *web* com Vue.js.

#### *3.2.2.1 Entrevista*

Esta etapa envolve entrevistar os especialistas, que respondem com os seus pontos de vista a respeito de questões relacionadas às melhores práticas no desenvolvimento de sistemas *web* com Vue.js. As perguntas são detalhadas a seguir.

1. O quanto você considera conhecer o *framework* Vue.js?
2. O quanto você considera conhecer sobre as boas práticas no uso de Vue.js?
3. Como você leva em consideração o seguimento de boas práticas quando você está desenvolvendo?
4. Se você se depara com um código que não está bem estruturado, você o modifica? Quando sim e quando não?
5. Quais são os fatores que levariam você a não seguir boas práticas no uso de Vue.js?

#### *3.2.2.2 Participantes*

Nesta ocasião, trata-se de um time que utiliza a metodologia ágil Scrum[21], composto por um gerente, um *Product Owner*, uma *Scrum Master* — que também é desenvolvedora — e outros seis indivíduos. Mais especificamente no projeto da interface do sistema, estão trabalhando quatro profissionais deste time: três especialistas de software e uma desenvolvedora de software.

[21] Scrum — Disponível em: <https://www.scrum.org>

Tabela 3.3 – Informações Sobre os Participantes

| Identificador | Escolaridade | Experiência Profissional | Cargo | Tempo no Projeto |
|---|---|---|---|---|
| P1 | Pós-graduação (Lato sensu) - Incompleto | 8 anos | Especialista de Software/Arquiteto | 4 meses |
| P2 | Superior - Incompleto | 9 anos | Especialista de Software | 4 meses |
| P3 | Superior - Completo | 14 anos | Especialista de Software | 2 meses |
| P4 | Pós-graduação (Lato sensu) - Incompleto | 3 anos | Desenvolvedora de Software/SM | 2 meses |

Fonte: Autora

Um dos especialistas é, inclusive, o arquiteto do time. Ele trabalha há mais de sete anos nesta empresa, lidando principalmente com questões de *back-end* e infraestrutura de TI. Outros dois especialistas são relativamente novos na empresa, mas trabalharam juntos em oportunidades passadas e, ainda, passaram por quatro diferentes empresas anteriormente, possuindo bons anos de experiência com TI, em particular nas áreas de *web design* e *front-end*. Por fim, há uma desenvolvedora *full stack* que, além disso, atua como *Scrum Master* do time há alguns meses.

## 3.3 Considerações Finais

Em síntese, usufruindo de um período de quatro meses, desde o início da prototipação até a data de entrega em ambiente de produção com, ao menos, as mais simples funcionalidades aos *stakeholders*, um sistema de catálogo de produtos e 14 informações a respeito destes foi desenvolvido. Esse sistema alvo foi implementado utilizando o *framework* Vue.js e algumas bibliotecas auxiliares, como Vuex e Vue Router.

Nesse contexto, há a oportunidade de analisar como é o uso de boas práticas na implementação do *front-end* em um projeto comercial de duas formas: quantitativa e qualitativa. Por meio da análise quantitativa, é possível encontrar possíveis descumprimentos de critérios de boas práticas de Vue.js e outras ferramentas manualmente, sugerindo possíveis melhores maneiras de implementá-las. E, por meio de entrevistas com os participantes envolvidos no projeto, é possível obter dados qualitativos dessa análise, a fim de entender o conhecimento do time sobre as tecnologias e suas boas práticas, além de conhecer suas rotinas de desenvolvimento de sistemas e possíveis adversidades nesse processo.

# 4 RESULTADOS DO ESTUDO

Neste capítulo, são apresentados os resultados obtidos a partir das análises realizadas. O estudo é baseado em um método de pesquisa misto, com base em dados quantitativos e qualitativos. Assim, pode-se não só identificar os descumprimentos de boas práticas das tecnologias adotadas, mas também complementar com o ponto de vista dos desenvolvedores a respeito desses fenômenos.

## 4.1 Dados Obtidos na Análise Quantitativa

O sistema alvo do presente trabalho trata de produtos e suas características. Assim, seguindo os procedimentos do estudo descritos no capítulo anterior, é possível analisar manualmente quantas violações de boas práticas das tecnologias utilizadas ocorreram, como é demonstrado na Tabela 4.1.

Tabela 4.1 – Total de Violações de Critérios de Boas Práticas

| **Critério** | **Total de Violações** |
|---|---|
| Convenções | 42 |
| Acoplamento | 42 |
| Padronização | 56 |
| *Lifecycle Hooks* | 14 |
| Arquivos de Declaração | 6 |
| Notação de Nomes de Componentes em Arquivos | 2 |
| Notação de Nomes de Propriedades em Arquivos | 0 |
| Notação de Nomes de Componentes em Templates | 0 |
| Notação de Nomes de Propriedades em Templates | 24 |
| Chave de Identificação no `v-for` | 1 |
| Não Uso de `v-if` com `v-for` | 0 |
| Estilos em Componentes com Escopo | 4 |
| Elementos Multiatributos | 5 |
| Aspas em Valores de Atributos | 0 |
| Abreviação de Diretivas | 2 |

Fonte: Autora

Figura 4.1 – Exemplo Inadequado dos Critérios Convenções e Acoplamento

```
// @/views/Product.vue
import { Product } from '@/constants';
import { ServiceLayer } from '@/services/product';

async insertProduct(newProduct: Product): void {
  await ServiceLayer.insertProduct(newProduct);
}
```

Fonte: Autora

### 4.1.1 Convenções, Acoplamento e Padronização

O padrão de gerenciamento de estado do sistema alvo é o Vuex, porém as ações de criação, edição e exclusão de 14 informações de produtos não estão usufruindo de *getters*, *mutations* e *actions*, o que totalizam 42 violações a respeito do critério **Convenções**. Visto que não há a possibilidade do Vuex ficar encarregado de gerenciar as ações, cada funcionalidade correspondente às ações de criação, edição e exclusão do sistema está chamando a camada de serviço diretamente, como é dado como exemplo na Figura 4.1 de uma função de inserir um novo produto. Portanto, também viola 42 vezes o critério **Acoplamento**. Essa situação gera um alto acoplamento nos componentes do sistema em questão, por haver dependências maiores que as necessárias. Por mais que o sistema utilize o Vuex, os componentes e as funcionalidades estão sendo obrigados a desencadear eventos entre si, tornando-os fortemente acoplados.

Nesta perspectiva, o objetivo é o *store* ficar encarregado dessas tarefas, seguindo o princípio *single source of truth*, e os componentes acionarem essas ações por meio de `this.$store.dispatch()`, do modo que é mostrado na Figura 4.2. Assim, a camada de *view* do sistema não se torna dependente da camada de serviço para funcionar e evita incoerências de dados em determinadas partes do software. É notável como a *store* pode facilitar a gestão de estado, pois ela passa a ser responsável por monitorar as mudanças da aplicação e notificar todos os componentes sobre novas atualizações de dados.

Ainda, levando em consideração as propriedades *getters*, *mutations* e *actions*, suas devidas constantes padronizadas também não foram desenvolvidas — estas apontadas pela Figura 4.3, o que causa 56 violações do critério **Padronização**, tratando-se de todas as ações CRUD do sistema, incluindo visualização.

Figura 4.2 – Exemplo Adequado dos Critérios Convenções e Acoplamento

```
// @/views/Product.vue
import { Product } from '@/constants';
import { INSERT_PRODUCT } from '@/store/action-types';

async insertProduct(newProduct: Product): void {
  await this.$store.dispatch('insertProduct', newProduct);
}

// @/store/modules/product.ts
import { ServiceLayer } from '@/services/product';
const product = new Vuex.Store({
  state: {
    currentProduct: { ... },
  },
  getters: {
    currentProduct: ({ currentProduct }): Product => {
      return currentProduct;
    },
  },
  mutations: {
    addProduct: (state, product: Product) => {
      state.currentProduct = { ...state.currentProduct, ...product };
    },
  },
  actions: {
    insertProduct: async ({ commit }, newProduct: Product) => {
      const response = await ServiceLayer.insertProduct(newProduct);
      commit('addProduct', response.data);
    },
  },
);

export default product;
```

Fonte: Autora

Figura 4.3 – Exemplo Adequado do Critério Padronização

```
// @/views/Product.vue
import { Product } from '@/constants';
import { INSERT_PRODUCT } from '@/store/action-types';

async insertProduct(newProduct: Product): void {
  await this.$store.dispatch(INSERT_PRODUCT, newProduct);
}

// @/store/getter-types.ts
export const CURRENT_PRODUCT = 'currentProduct';

// @/store/mutation-types.ts
export const ADD_PRODUCT = 'addProduct';

// @/store/action-types.ts
export const INSERT_PRODUCT = 'insertProduct';

// @/store/modules/product.ts
import { ServiceLayer } from '@/services/product';
import { CURRENT_PRODUCT } from '@/store/getter-types';
import { ADD_PRODUCT } from '@/store/mutation-types';
import { INSERT_PRODUCT } from '@/store/action-types';

const product = new Vuex.Store({
  state: {
    currentProduct: { ... },
  },
  getters: {
    [CURRENT_PRODUCT]: ({ currentProduct }): Product => {
      return currentProduct;
    },
  },
  mutations: {
    [ADD_PRODUCT]: (state, product: Product) => {
      state.currentProduct = { ...state.currentProduct, ...product };
    },
  },
  actions: {
    [INSERT_PRODUCT]: async ({ commit }, newProduct: Product) => {
      const response = await ServiceLayer.insertProduct(newProduct);
      commit(ADD_PRODUCT, response.data);
    },
  },
);

export default product;
```

Fonte: Autora

Figura 4.4 – Exemplo Inadequado do Critério *Lifecycle Hooks*

```
// @/views/Product.vue
import { ServiceLayer } from '@/services/product';

async created() {
  this.product = await ServiceLayer.loadProduct();
  this.productInformation = await ServiceLayer.loadProductInfo();
}

// @/router/index.ts
import Product from '@/views/Product.vue';
import { RouteConfig } from 'vue-router';

export default const routes: RouteConfig[] = [{
  path: '/product/',
  name: 'product',
  component: Product,
}];
```

Fonte: Autora

### 4.1.2 *Lifecycle Hooks*

Neste projeto, é fundamental o carregamento apropriado de informações relacionadas ao produto para que o usuário possa navegar pelo sistema *web* adequadamente, tratando-se da funcionalidade de visualização no que diz respeito ao CRUD. Com este fim, foram implementadas 14 chamadas de *data fetching* no *hook* (gatilho, em inglês) de ciclo de vida `created()`, que executa código logo após a instância Vue ser criada, violando o critério ***Lifecycle Hooks***.

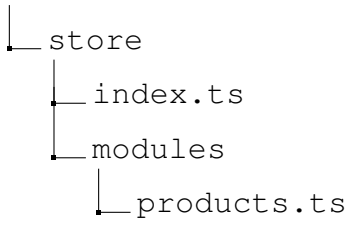
Porém, o ideal é chamar antes desta etapa, para que se execute lógicas personalizadas caso seja necessário, sem prejudicar a experiência do usuário. Para isso, é preciso utilizar o *navigation guard* `beforeEnter()`, como é apresentado na Figura 4.5.

Figura 4.5 – Exemplo Adequado do Critério *Lifecycle Hooks*

```
// @/router/guards.ts
import store from '@/store';
import { NavigationGuard } from 'vue-router';
import { LOAD_PRODUCT, LOAD_PRODUCT_INFO } from '@/store/action-types';

export const loadProductInformation: NavigationGuard = async (
  to, from, next,
) => {
  await store.dispatch(LOAD_PRODUCT, to.params.id);
  await store.dispatch(LOAD_PRODUCT_INFO);
  next();
};

// @/router/index.ts
import Product from '@/views/Product.vue';
import { RouteConfig } from 'vue-router';
import { loadProductInformation } from './guards';

export default const routes: RouteConfig[] = [{
  path: '/product/',
  name: 'product',
  component: Product,
  beforeEnter: loadProductInformation,
}];
```

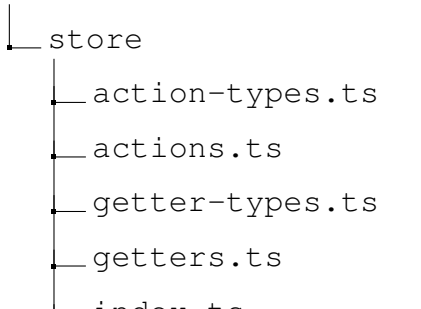
Fonte: Autora

Figura 4.6 – Exemplo Inadequado do Critério Arquivos de Declaração




Fonte: Autora

Figura 4.7 – Exemplo Adequado do Critério Arquivos de Declaração




Fonte: Autora

### 4.1.3 Arquivos de Declaração

Visto que há violações no uso dos padrões de Vuex, em que não foram implementados *getters*, *mutations* e *actions* devidamente, não foi notada a necessidade de separar o conteúdo do arquivo do *store*, e a estrutura do projeto é semelhante ao exemplo da Figura 4.6.

No entanto, é interessante fazer estas separações em múltiplos arquivos assim que as convenções do Vuex começarem a ser empregadas no sistema, já que tornará o arquivo extenso e comprometerá a manutenibilidade se mantê-las em apenas um arquivo. O ideal é criar três arquivos para cada uma das funcionalidades do Vuex: `actions.ts`, `getters.ts` e `mutations.ts`, assim como três arquivos para separar as constantes a serem criadas e utilizadas por estas funcionalidades: `action-types.ts`, `getter-types.ts` e `mutation-types.ts`, como indica a Figura 4.7. Isto posto, há seis violações do critério **Arquivos de Declaração**, referente aos seis arquivos que não foram criados.

### 4.1.4 Notação de Nomes

Com relação aos padrões referente às notações de nomes de componentes e propriedades da aplicação, foram analisados:

- Componentes em Arquivos: As convenções de nomes multipalavras e notação com `PascalCase`, cinco dos componentes possuem nomes compostos de mais de uma palavra, menos os componentes `Home` e `Product` que não se encontram de acordo com a convenção.
- Propriedades em Arquivos: A convenção de notação com `camelCase`, todas as declarações de propriedades em arquivos estão de acordo.
- Componentes em Templates: As convenções de nomes multipalavras e notação com `kebab-case`, todas as declarações de componentes em templates estão de acordo.
- Propriedades em Templates: A convenção de notação com `kebab-case`, as nove propriedades são usadas 42 vezes ao longo do código, e 24 vezes elas estão sendo chamadas com `camelCase` ao invés de `kebab-case`, não correspondendo ao padrão definido.

Neste sentido, tanto o critério **Notação de Nomes de Componentes em Arquivos** quanto o critério **Notação de Nomes de Propriedades em Templates**, possuem violações. Estas tendem a gerar conflitos no que diz respeito ao uso de componentes e propriedades nas interfaces do sistema, já que o HTML é insensível a maiúsculas e minúsculas e os seus elementos fornecidos são formados por apenas uma única palavra.

### 4.1.5 Uso de `v-for`

No quesito de uso de `v-for`, totalizando quatro aparições, a `:key` não é implementada corretamente uma única vez, fazendo com que esta listagem em questão não garanta uma ordenação previsível, prejudicando o estado interno do componente e violando uma vez o critério **Chave de Identificação no `v-for`**. Felizmente, em nenhum caso de `v-for` o `v-if` é usado, o que evita verificações desnecessárias no *loop* das listagens e o critério **Não Uso de `v-if` com `v-for`** não é violado.

Figura 4.8 – Exemplo Inadequado do Critério Abreviações de Diretivas

```
template(v-slot:before="{item}")
template(v-slot:after="{item}")
```

Fonte: Autora

Figura 4.9 – Exemplo Adequado do Critério Abreviações de Diretivas

```
template(#before="{item}")
template(#after="{item}")
```

Fonte: Autora

### 4.1.6 CSS

Dentre os sete componentes, quatro não aplicam `scoped` à tag *style*, que trata do CSS dos componentes, sendo preciso criar classes de estilo com nomes muito específicos para eles. Isto é necessário para que não haja conflitos em outros lugares do sistema, o que não facilita o desenvolvimento de estilos dos componentes e viola quatro vezes o critério **Estilos em Componentes com Escopo**.

### 4.1.7 HTML

Ao longo do código, é visto um total de 44 elementos HTML que possuem quaisquer atributos sendo passados como parâmetro, todos eles usando aspas em seus valores, e 34 destes elementos possuem mais de um atributo. No entanto, cinco dos elementos não estavam quebrando seus atributos em mais de uma linha, violando cinco vezes o critério **Elementos Multiatributos**. Para melhorar a legibilidade do código, o ideal é separar cada atributo em uma linha. Além disso, os trechos de código nos quais usufruem das diretivas `v-on` e `v-bind` estavam de acordo com suas abreviações (`@` e `:`, respectivamente). Porém, para casos de uso da diretiva `v-slot`, que foram implementadas duas vezes, não está sendo utilizada a sua abreviação, como mostra a Figura 4.8, o que viola duas vezes o critério **Abreviação de Diretivas**. Logo, com o intuito de manter um padrão para todo o sistema, deve-se utilizar a abreviação `#`, como indica a Figura 4.9.

## 4.2 Conclusões Finais da Análise Quantitativa

Considerando os detalhes do sistema alvo, em que 14 informações relacionadas a produtos estão sendo implementadas com as respetivas funções CRUD, é observado um total de 198 violações de acordo com 11 dos 15 critérios selecionados na avaliação, concentrando-se principalmente os primeiros três critérios relacionados aos conceitos básicos do gerenciamento de estado Vuex.

A falta de *getters*, *mutations* e *actions* no sistema desencadeou um total de 140 violações para os critérios **Convenções**, **Acoplamento** e **Padronização**, correspondendo a uma maneira não ideal do uso do Vuex em um sistema *web* que emprega o *framework* Vue.js, afetando sua qualidade no quesito de possuir uma base solida padronizada para funcionar adequadamente. Ainda nesse cenário, é resultado um alto acoplamento, visto que a camada *view* da aplicação precisa chamar a camada de serviço para qualquer ação de criação, edição e exclusão realizada pelo usuário.

Não obstante, o modo como a funcionalidade de visualização de dados está sendo implementada também não é pertinente: as 14 chamadas de *data fetching* estão acontecendo sincronicamente após a instância Vue ser criada, o que viola boas práticas do critério de ***Lifecycle Hooks***, já que não é possível lidar da melhor maneira com os resultados de todos os processamentos que são feitos, como a possibilidade de manusear erros e alertas diversos satisfatoriamente.

Houveram, ainda, alguns problemas com relação aos arquivos da aplicação, que ocasionaram 6 violações ao critério **Arquivos de Declaração**, que compromete a manutenibilidade ao não separar em múltiplos arquivos as funcionalidades do sistema, e 2 violações ao critério **Notação de Nomes de Componentes em Arquivos** por não criar todos os componentes com nome composto, o que deixa de prevenir possíveis conflitos com elementos HTML quando usados nos templates do sistema.

Seguindo com a ideia da última colocação a respeito de templates, questões de violações às boas práticas da organização de HTML e CSS também foram notadas, tratando-se do restante das 36 violações dispersas em cinco outros critérios. Percebe-se, de modo geral, questões de falta de padrão por conta de implementações intermitentes das boas práticas, que por vezes é seguida e por vezes não é.

## 4.3 Dados Obtidos na Análise Qualitativa

Com relação aos dados qualitativos deste trabalho, é descrito o entendimento dos relatos dados de três especialistas que participaram do desenvolvimento do sistema alvo. Estas informações foram coletadas por meio de cinco perguntas definidas no capítulo anterior, seguido pela sua leitura e relações entre si. As informações são organizadas de acordo com os tópicos abordados ao longo de suas respostas.

### 4.3.1 Conhecimento sobre Vue.js

No decorrer da explicação dos participantes sobre o entendimento de Vue.js, eles demonstraram estar familiarizados com o *framework*, porém reconhecem que não possuem conhecimento avançado sobre ele. Mesmo que trabalhem com a linguagem de programação JavaScript há muitos anos, o Vue.js é uma tecnologia que usam há poucos meses, e existem detalhes de implementações que podem não ter conhecimento ainda.

Diante disso, ao refletirem sobre situações em que há pouco conhecimento sobre qual a melhor maneira de desenvolvimento, a documentação é usada como uma das estratégias para quando surgem estas dúvidas ao longo do processo.

> [...] Aquilo que eu não sei, eu procuro na documentação. Conheço a documentação, conheço onde procurar, então aqueles detalhes que a gente não sabe ou que não usa todo dia, no momento que a gente precisa usar, a gente acaba procurando na documentação. [P3]

Ademais, há uma certa preocupação quanto à nova versão do *framework*, lançada em 18 de setembro de 2020[1], em que são apresentadas novas maneiras de implementação de funcionalidades. Nesses casos, o conhecimento sobre Vue.js precisa ser atualizado para estar de acordo com as novidades.

> [...] Faz pouco tempo, surgiu a nova versão do Vue, que é o Vue 3, e com a versão 3 eu ainda não tive experiência, só li algumas coisas a respeito, então talvez tem que considerar que, com a versão nova, o conhecimento tem que melhorar um pouquinho. [P3]

Dessa forma, os três se consideram como um nível intermediário de entendimento, uma vez que entendem bem de arquitetura de software e aprenderam ao longo do tempo como funciona o fluxo de dados de aplicações implementadas com o *framework* e suas bibliotecas auxiliares, mas existirem particularidades que desconhecem.

[1] Vue.js v3.0.0 "One Piece" Released! — Disponível em <https://news.vuejs.org/issues/186>

### 4.3.2 Seguimento de Boas Práticas

As boas práticas são critérios da fase de desenvolvimento para se seguir, e estes são triviais pelos entrevistados, que se sentem confiantes neste quesito em razão de, habitualmente, estarem acostumados a pesquisar e estudar as mais favoráveis maneiras de se implementar sistemas em Vue.js. Porém, também identificam que não dispõem de conhecimento avançado, em consequência de se tratar de uma tecnologia que possui suas próprias particularidades, necessitando de estudos específicos além do conhecimento prévio em JavaScript que já possuem.

No decorrer das respostas, são levantadas algumas estratégias para adquirir conhecimento sobre as boas práticas e garantir a qualidade do código daquilo melhores é desenvolvido, como uso de *linters*, cursos online e consultas ao Vue.js *Style Guide*, guia de estilo essencial disponibilizado pelo próprio *framework*, diferenciando-se de outros *frameworks* JavaScript disponíveis no mercado de trabalho. Contudo, segundo a própria referência do *Style Guide*, existem alguns usos de boas práticas em que é preciso optar por um padrão ou por outro. Deste modo, é previsto acordos entre todos os envolvidos do time para estas tomadas de decisão, em que estabelecem critérios que tenham sentido e que tragam valor, tornando-se fatores de referência de padronização para futuros projetos.

> [...] É sempre discutindo em conjunto com o time, nunca é algo determinado, sempre procuramos discutir com o time pra ver se todos concordam que aquela forma que nós estamos fazendo é a melhor forma. [P3]

> [...] Conforme vamos aprendendo e vamos nos aprofundando nesses novos conceitos, diversas melhorias são feitas. É desafiador, mas vale a pena, pois um código organizado sempre gera menos retrabalho e conseguimos sentir as vantagens em nossa rotina. [P2]

Sendo assim, as melhores práticas são muito consideradas pelos especialistas durante o desenvolvimento de um sistema com Vue.js, em virtude de concordarem com a importância delas para a qualidade do software e seus diversos outros benefícios como, por exemplo, o aprimoramento da legibilidade do código.

> [...] O mais importante, o principal, é o código ser legível, "limpo", estar testável, que faça sentido com aquele paradigma; outra pessoa tem que conseguir ler/entender o código. [P1]

Porém, adversidades são identificadas pelos integrantes do time, independentemente do tempo de experiência profissional. Assim, apesar de tentarem adotar as práticas sempre que possível, podem surgir certos desafios. Alguns contratempos são percebidos por conta da tecnologia ser volátil, ou seja, estar sempre se atualizando com o passar o tempo. Consequentemente, ocorrem situações em que confirmam que podem não estar

totalmente seguros quanto ao seguimento das melhores maneiras de desenvolvimento. Há aborrecimentos, por exemplo, sobre falta de auxílio disponível na internet sobre o uso adequado de Vue.js com TypeScript, tipagem estática adotada para o desenvolvimento do sistema alvo. Mas se mostram dispostos para agir em prol de um código melhorado.

> [...] Se a gente descobrir, no futuro, uma melhor forma de arquitetar, de usar alguma funcionalidade, a gente vai, com certeza, rediscutir e ver se é a melhor forma ou não. [P3]

### 4.3.3 Violações de Boas Práticas

A equipe admite que há momentos que não seguem as boas práticas do Vue.js, embora considerem ser casos pontuais, exceções que não são recorrentes. Algumas razões nas quais levariam os participantes a não seguir certos critérios são situações em que é preciso validar uma prova de conceito (POC) ou entregar um produto mínimo viável (MVP). Mas é preciso estar atento a este tipo de abordagem, uma vez que pode se tornar um modo ligeiro de agir para entregar demandas e atender objetivos a curto prazo, o que prejudicaria a qualidade do produto final.

> [...] O que pode acontecer é que, em casos específicos, seja necessário entregar algo transitório, provisório, que não é a versão final. Entregar algo que dê margem pra outro código que venha depois, que evolua para um produto, uma aplicação mais durável. [P1]

Ainda, é possível notar problemas quanto aos requisitos, regras de negócio e prazos definidos no momento dos acordos entre os *stakeholders* e a equipe, que podem mudar a todo o momento. Nesse cenário, os envolvidos no projeto são impactados pela falta de definições, fazendo com que exista uma falta de necessidade em se seguir padrões.

> [...] Às vezes acontece tantos *requests* de *business*, que mudam o tempo todo, sem nada definitivo, que não vale a pena perder tanto tempo modelando, fazendo testes, etc. Funciona bem quando tu "tem" *requirements* prontos, quase imutáveis. [P1]

Outro fator importante lembrado é em momentos quando há pouquíssimo ou nenhum conhecimento sobre as diversas melhores práticas. Isso ocorre principalmente nas fases de aprendizado, em condições nas quais as tecnologias são conhecidas há pouco tempo pelos desenvolvedores. Em tais casos, é sugerido pelos entrevistados ter proatividade em buscar esse conhecimento e, ainda, questionar as técnicas e abordagens de desenvolvimento que o time está acostumado, a fim de descobrir maneiras mais prudentes e benéficas de realizar as atividades.

As circunstâncias citadas pelos entrevistados vieram a acontecer durante o desenvolvimento do sistema alvo estudado, o que fez com que ocasionasse tantas violações aos critérios de boas práticas, estas vistas na análise quantitativa deste trabalho.

> [...] Quando começamos o projeto, tudo era novidade para todos. Faltavam referências, não sabíamos por onde começar. [P2]

### 4.3.4 Manutenção de Código Fonte

Quando os especialistas entrevistados se deparam com falhas no sistema, ou seja, quando encontram blocos de código mal estruturados, eles alegam que existem situações apropriadas e outras que não são para modificá-los e melhorá-los. Há circunstâncias em que é preferível não realizar grandes mudanças, principalmente por conta de um forte acoplamento existente no sistema.

> [...] Se é um código maior que vá impactar em outros lugares, provavelmente eu não vou modificar. [P1]

Por outras vezes, uma modificação pode exigir tempo e esforço em excesso dos desenvolvedores, e fazer com que sejam desviados do foco principal de uma determinada tarefa definida previamente. Em casos mais complexos, é preciso uma grande refatoração, que não vai bastar modificar rapidamente apenas um único bloco de código. Em qualquer caso, é concordado que é sempre preciso comunicar a equipe sobre os problemas encontrados, para mapear e definir momentos mais propícios para executar tarefas de melhoria e evolução do código.

> [...] De qualquer forma, antes de tomar qualquer ação, devemos deixar a equipe à par da situação. Aí teremos um mapeamento mais assertivo dos pontos de melhoria ou de atenção. [P2]

> [...] O que eu vou fazer é solicitar uma mudança estrutural daquele código, uma *user story*, uma requisição pro time para rever aquele código como um todo: vira um trabalho de *refactoring*. [P1]

Por fim, o time afirma e concorda sobre os benefícios de um software com código bem estruturado, de fácil manutenibilidade e que segue as boas práticas, para que se ele continue útil e relevante por muito tempo, mesmo que isso exija mais esforço dos programadores.

> [...] Uma das coisas que aprendi, e trago como regra nesses anos de carreira, é sempre fazer o melhor que estiver ao alcance... É muito mais comum nos arrependermos por algum detalhe que deixamos pra trás, do que pelo tempo extra investido para aprimorar alguma lógica. [P2]

## 4.4 Conclusões Finais da Análise Qualitativa

Com base nas respostas dos entrevistados, observa-se que os especialistas de software possuem conhecimento intermediário sobre o *framework* Vue.js. Eles não se consideram novatos, em consequência de estarem trabalhando há alguns meses com as tecnologias empregadas e, também, possuírem vasta experiência na área de desenvolvimento *web*.

Sempre que possível, os envolvidos no projeto buscam seguir as melhores práticas das tecnologias empregadas nos sistemas que estão desenvolvendo, visto que entendem a importância delas para garantir uma melhor qualidade de código fonte. Porém, concordam que há algumas particularidades e recomendações que não possuem total domínio a respeito dessas boas práticas de Vue.js e suas bibliotecas auxiliares.

Não obstante, possíveis complicações e obstáculos costumam surgir em meio aos processos e atividades do ciclo de vida de software, o que compromete a qualidade do que é feito. Alguns destes problemas são destacados, como entrega de demandas apressadas sem os devidos cuidados, falta de definições adequadas sobre os requisitos do sistema e pouco ou nenhuma noção sobre boas práticas, sobretudo em fases de aquisição de conhecimento sobre novas tecnologias. Por estas razões, algumas violações das boas práticas foram feitas no sistema alvo analisado, sendo preciso arcar com retrabalhos de manutenção futuros em busca de uma melhoria contínua deste código fonte, impactando na produtividade do time.

Nesse contexto, o time precisa agir e entrar em acordos de padronizações fundamentadas e níveis de aptidão mínimos, com o propósito de sofisticar os processos em que estão inseridos e, também, de conseguirem pesquisar profundamente sobre as ferramentas e técnicas utilizadas, através de cursos e leituras cautelosas nas documentações disponibilizadas na internet.

# 5 CONCLUSÃO

O presente trabalho teve como proposta analisar a qualidade do código fonte de um sistema *web*, tratando-se de um catálogo centralizador de produtos e 14 informações a respeito destes. Foram observados 198 descumprimentos dos 11 dos 15 critérios de boas práticas selecionados na avaliação. Estes critérios se referem ao uso do *framework* Vue.js e tecnologias auxiliares, como Vuex, Vue Router e TypeScript, levando em consideração suas documentações e o Vue *Style Guide*.

A partir dos resultados obtidos, foram descritas maneiras de aperfeiçoamento do código estudado, de modo a trazer mais garantia de qualidade e padronização naquilo que é desenvolvido com as ferramentas propostas, sendo inovador no quesito de análise de qualidade de código fonte com Vue.js.

Além disso, os especialistas de software envolvidos no desenvolvimento do sistema alvo analisado foram entrevistados, a fim de coletar informações a respeito de suas opiniões sobre o conhecimento e confiança sobre o uso do *framework* Vue.js e as suas boas práticas, assim como mais detalhes de suas rotinas como programadores. Foi observado que estes participantes da entrevista realizada se encontram, atualmente, familiarizados quanto ao uso de Vue.js e suas boas práticas, e compreendem a relevância de seguir convenções definidas previamente e suas vantagens.

Por vezes, alguns desafios e impedimentos são reconhecidos no dia a dia, como entregas de valor em curto prazo, regras de negócio que são modificadas a todo o momento e falta de conhecimento das práticas adotadas, que fazem com que nem sempre seja possível realizar as implementações com maestria. Foram estes fatores que influenciaram no descumprimento dos critérios de boas práticas analisados no sistema alvo, que foi desenvolvido num período de quatro meses e será preciso ser retrabalhado por mais um período de tempo após a entrega em ambiente de produção aos usuários finais, a fim de realizar planos de melhoria fundamentados a longo prazo.

Mesmo que princípios relacionados à qualidade de software, tratando-se tanto de fatores de qualidade internos, como a manutenibilidade de um sistema, quanto fatores de qualidade externos, de garantir que o produto a ser entregue está de acordo com as necessidades dos usuários finais, sejam referidos como uma prioridade e com seriedade, podem ocorrer situações que prejudicam os processos de ciclo de vida de um software, e podem gerar problemas de manutenção e evolução de código fonte. Para que seja possível reverter tais cenários, é fundamental que exista uma boa comunicação e proatividade

entre todos os integrantes do time e que, a partir do trabalho em equipe, ajam em prol de avanços na metodologia das atividades desenvolvidas, mapeando pontos de possíveis aperfeiçoamentos.

Por fim, este estudo abre oportunidade para alguns trabalhos futuros, entre eles a investigação sobre possíveis novas melhores práticas que venham a surgir com a terceira versão do *framework* Vue.js, lançada em setembro de 2020, em conjunto do uso adequado de tipagem estática com a tecnologia TypeScript. Poderiam, também, ser executados estudos de avaliação na perspectiva dos *stakeholders*, por meio de *feedback* dos seus pontos de vista sobre a eficiência, usabilidade e satisfação acerca do sistema alvo como um todo, com o objetivo de analisar fatores de qualidade externos.

# REFERÊNCIAS

AGARWAL, B.; TAYAL, S. **Software Engineering**. [S.l.]: Laxmi Publications Pvt Limited, 2009. ISBN 9788190855914.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. [S.l.]: Pearson Education, 2012. (SEI Series in Software Engineering). ISBN 9780132942782.

GRADY, R. B.; CASWELL, D. L. **Software Metrics: Establishing a Company-wide Program**. [S.l.]: Prentice-Hall, 1987. ISBN 9780138218447.

HANCHETT, E.; LISTWON, B. **Vue.js in Action**. 1st. ed. USA: Manning Publications Co., 2018. ISBN 1617294624.

HUNT, A.; THOMAS, D. **The Pragmatic Programmer: From Journeyman to Master**. Harlow, England: Addison-Wesley, 1999. ISBN 978-0-201-61622-4.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. Piscataway, US, 1990. 1-84 p.

ISO/IEC JTC 1/SC 7. **Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**. Geneva, CH, 2011. Accessed August 2020. Available from Internet: <https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en>.

ISO/IEC JTC 1/SC 7. **Systems and software engineering — Vocabulary**. Geneva, CH, 2017. Accessed August 2020. Available from Internet: <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>.

KRUCHTEN, P.; NORD, R.; OZKAYA, I. **Managing Technical Debt: Reducing Friction in Software Development**. [S.l.]: Pearson Education, 2019. (SEI Series in Software Engineering). ISBN 9780135645963.

LEHMAN, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. **Journal of Systems and Software**, Elsevier Science Inc., USA, v. 1, p. 213–221, sep. 1980. ISSN 0164-1212.

MACRAE, C. **Vue.Js: Up and Running Building Accessible and Performant Web Apps**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2018. ISBN 1491997192.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River, NJ: Pearson Education, 2008. (Robert C. Martin Series). ISBN 9780136083252.

Ocariza, F. et al. An empirical study of client-side javascript bugs. In: **2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2013. p. 55–64.

Ocariza, F. S. et al. A study of causes and consequences of client-side javascript bugs. **IEEE Transactions on Software Engineering**, v. 43, n. 2, p. 128–144, 2017.

Ocariza Jr., F. S.; Pattabiraman, K.; Zorn, B. Javascript errors in the wild: An empirical study. In: **2011 IEEE 22nd International Symposium on Software Reliability Engineering**. [S.l.: s.n.], 2011. p. 100–109.

Saboury, A. et al. An empirical study of code smells in javascript projects. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 294–305.

SAMETINGER, J. **Software Engineering with Reusable Components**. [S.l.]: Springer Berlin Heidelberg, 2013. ISBN 9783662033456.

WOHLGETHAN, E. **Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js**. Bachelor's Thesis — IT department, Haw Hamburg, 2018.

YOU, E. **Guide - Vue.js**. 2017. Accessed August 2020. Available from Internet: <https://vuejs.org/v2/guide/>.

YOURDON, E. **Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible"Projects**. 2nd. ed. [S.l.]: Prentice Hall Professional Technical Reference, 2004. (Software engineering). ISBN 9780131436350.

# APÊNDICE A — ENTREVISTA COM PARTICIPANTE 1

1. **O quanto você considera conhecer o *framework* Vue.js?**
   Eu tenho um conhecimento médio. Eu já não sou mais básico, eu sei como é que funciona o *framework*, como usar... Como usar o *store*, como o fluxo de dados funciona dentro dele... Mas detalhes muito mais avançados eu não sei.
2. **O quanto você considera conhecer sobre as boas práticas no uso de Vue.js?**
   Eu colocaria como médio também. Não só por Vue.js, mas pelo JavaScript no geral, né. Eu me consideraria um cara avançado no JavaScript "puro", então eu tenho uma noção boa do que deveria ser feito no Vue.js. Não me coloco avançado (em Vue.js) porque sempre muda várias coisas, dependendo da versão pode ter recomendações diferentes.
3. **Como você leva em consideração o seguimento de boas práticas quando você está desenvolvendo?**
   Sempre busco as boas práticas, inclusive com o uso de ferramentas de *lint/format* de código. Mas não me considero um cara "tudo ou nada". Tem vezes que há exceções, mas não são recorrentes. Eu considero que boas práticas são muito importantes não só para manter o código conciso, mas também capacitar a outras pessoas poderem ler o código depois, ou até o próprio desenvolvedor mesmo poder ler mais tarde e lembrar como que faz as coisas. O mais importante, o principal, é o código ser legível, "limpo", estar testável, que faça sentido com aquele paradigma; outra pessoa tem que conseguir ler/entender o código.
4. **Se você se depara com um código que não está bem estruturado, você o modifica? Quando sim e quando não?**
   Isso depende... Se é pouco código, eu sempre modifico. Agora, se é um código maior que vá impactar em outros lugares, provavelmente eu não vou modificar. Eu posso até modificar alguma parte que eu consiga simplificar o meu trabalho, mas o que eu vou fazer é solicitar uma mudança estrutural daquele código, uma *user story*, uma requisição pro time para rever aquele código como um todo: vira um trabalho de *refactoring*.
5. **Quais são os fatores que levariam você a não seguir boas práticas no uso de Vue.js?**
   As boas práticas sempre fizeram sentido, principalmente no caso as que eu conheço do Vue.js. O que pode acontecer é que, em casos específicos, seja necessário en-

tregar algo transitório, provisório, que não é a versão final. Entregar algo que dê margem pra outro código que venha depois, que evolua para um produto, uma aplicação mais durável... Aí eu não respeitaria as boas práticas. Agora, se é um código que a intenção é que ele dure, que ele seja pra sempre, né? Aí não tem desculpas, aí é seguir as boas práticas do Vue.js. Em geral, vou sempre seguir as regras... Bom, eu não deixaria de usá-las conhecendo elas, mas obviamente se eu não conhecer (as boas práticas) eu não usarei elas.

Uma outra situação é se, pra usar uma boa prática, tu vai precisar mexer em muito código que vai além do que tu precisa pra resolver um problema. Talvez se eu conseguir encapsular uma boa prática num pedaço, e reaproveitar, melhor. Mas, se não, eu vou fazer do jeito que dá, mas vou alertar o time para que isso seja resolvido "lá na frente". O importante é mapear o que tá errado, o que tá ruim, o que precisa de melhoria e deixar pra depois, pra priorizar depois. Senão, tu entra em vários "probleminhas" que existem e nem consegue entregar o que era previsto do teu código, que não tem nada a ver, e isso é bem comum.

Outra situação é a respeito de quando há falta de definições, com mudanças muito frequentes... Acabam impactando. Às vezes acontece tantos *requests* de *business*, que mudam o tempo todo, sem nada definitivo, que não vale a pena perder tanto tempo modelando, fazendo testes, etc. Funciona bem quando tu "tem" *requirements* prontos, quase imutáveis. Ocorre também uma relação com os prazos definidos aí. Assim, é entregue um projeto que funciona mas com funcionalidades que não estão tão boas e que precisarão ser revisitadas, com testes que não foram feitos.

Além disso, falta conhecimento sobre as boas práticas nas pessoas, não seguem porque não conhecem mesmo, e às vezes não vão atrás. Conhecimento de boas práticas de desenvolvimento e linguagens com *frameworks* diferentes também. A pessoa tem que ter um bom senso de se perguntar "só tem esse jeito de fazer? Não tem outra forma melhor?"... É bom ler guias para cada linguagem, cada *framework*.

## APÊNDICE B — ENTREVISTA COM PARTICIPANTE 2

1. **O quanto você considera conhecer o *framework* Vue.js?**
   Trabalho com JavaScript desde o início da minha carreira. Ao longo desses anos pude me aprofundar na linguagem e seus principais *frameworks* e adquirir uma sólida experiência em arquitetura de software. Então, mesmo desenvolvendo a poucos meses com Vue.js, já fui exposto a diversas situações nas quais aprendi bastante.
2. **O quanto você considera conhecer sobre as boas práticas no uso de Vue.js?**
   Apesar de ser meu primeiro contato, quando o assunto é tecnologia, conhecimento sempre é algo muito volátil... Independente do nível de experiência com o *framework*, algumas convenções são vinculadas a linguagem, mas sempre surgem desafios que ensinam e revalidam diversos conceitos... Mas, no contexto do meu trabalho, posso dizer que me sinto bastante seguro.
3. **Como você leva em consideração o seguimento de boas práticas quando você está desenvolvendo?**
   Uma das coisas que aprendi, e trago como regra nesses anos de carreira, é sempre fazer o melhor que estiver ao alcance... É muito mais comum nos arrependermos por algum detalhe que deixamos pra trás, do que pelo tempo extra investido para aprimorar alguma lógica
4. **Se você se depara com um código que não está bem estruturado, você o modifica? Quando sim e quando não?**
   Confesso que depende muito do quanto precisarei dele na implementação atual ou de sua complexidade... Há casos e casos. Em alguns é menos trabalhoso compreender a lógica e reescrever tudo, pois há poucos casos que basta uma pequena readequação. Mas, infelizmente, na maioria das vezes a implementação não é tão clara, então é melhor evitar problemas que uma refatoração imatura pode trazer. De qualquer forma, antes de tomar qualquer ação, devemos deixar a equipe à par da situação. Aí teremos um mapeamento mais assertivo dos pontos de melhoria ou de atenção.
5. **Quais são os fatores que levariam você a não seguir boas práticas no uso de Vue.js?**
   A única justificativa para não seguir boas práticas, na minha humilde opinião, é a falta de conhecimento. E, quando começamos o projeto, tudo era novidade para todos. Faltavam referências, não sabíamos por onde começar. Mas, ainda assim,

conforme vamos aprendendo e vamos nos aprofundando nesses novos conceitos, diversas melhorias são feitas. É desafiador, mas vale a pena, pois um código organizado sempre gera menos retrabalho e conseguimos sentir as vantagens em nossa rotina. Uma arquitetura bem fundamentada nos garante manutenibilidade, independente da tecnologia, escopo ou prazo. Aliás, já testemunhei algumas provas de conceito sendo usadas como base para implementações maiores, entre outros, mas nunca sabemos as proporções que um sistema pode tomar. Por isso, quanto antes dermos atenção aos detalhes, melhor.

# APÊNDICE C — ENTREVISTA COM PARTICIPANTE 3

1. **O quanto você considera conhecer o *framework* Vue.js?**

   Eu acho que eu conheço bem. Não conheço tudo, obviamente, pequenos detalhes, coisas mais específicas que a gente não usa todo dia, talvez não conheça algumas. Mas, em geral, eu diria que eu conheço bem, e aquilo que eu não sei, eu procuro na documentação. Conheço a documentação, conheço onde procurar, então aqueles detalhes que a gente não sabe ou que não usa todo dia, no momento que a gente precisa usar, a gente acaba procurando na documentação.

   Faz pouco tempo, surgiu a nova versão do Vue, que é o Vue 3, e com a versão 3 eu ainda não tive experiência, só li algumas coisas a respeito, então talvez tem que considerar que, com a versão nova, o conhecimento tem que melhorar um pouquinho... Eu tenho que aprender principalmente como é que funciona a *Composition API*, que é uma espécie de API que eles criaram com métodos que vai substituir os *mixins*, mas eu considero ainda que o conhecimento é bom, no Vue, mesmo não tendo trabalhado ainda com a versão 3.

2. **O quanto você considera conhecer sobre as boas práticas no uso de Vue.js?**

   Eu acho que eu conheço bastante também, por esse tempo de experiência trabalhando com Vue. Também por ter sempre estudado o que eu vou fazer antes de fazer, na verdade, então, por ter assistido alguns cursos de Vue para tirar minhas dúvidas e também aprender sobre essas boas práticas. E Vue também tem uma particularidade que, acredito que o React não tenha ainda: tem um *Style Guide* desenvolvido pelo próprio time que desenvolve o Vue, na própria página do Vue, e esse *Style Guide* é um manual de boas práticas que devem se fazer, e a prioridade dessas coisas, se é uma coisa essencial que deve se fazer sempre, que "deve" se fazer, que "pode" fazer, enfim, eles tem ali a descrição e a orientação de como deve criar determinadas coisas, como *props*.

   Então, eu acho que meu conhecimento das boas práticas do Vue.js vem da experiência e de consultar este manual quando necessário. E, também, da nossa experiência em particular, principalmente que usamos TypeScript... Pro Vue.js 2 tem pouca documentação de como usar ele com TypeScript, então muitas das coisas que a gente usa foram discutidas e se chegou numa conclusão em conjunto de que aquilo ali, até o momento, é a melhor forma de fazer. Então, o que a gente usa hoje em dia é baseada nessas três coisas: *Style Guide*, experiência anterior e experiência nova,

com aplicações usando TypeScript, e é sempre discutindo em conjunto com o time, nunca é algo determinado, sempre procuramos discutir com o time pra ver se todos concordam que aquela forma que nós estamos fazendo é a melhor forma. E não significa que a maneira como qual a gente fez, nesse momento, é a melhor de todas: se a gente descobrir, no futuro, uma melhor forma de arquitetar, de usar alguma funcionalidade, a gente vai, com certeza, rediscutir e ver se é a melhor forma ou não.

3. **Como você leva em consideração o seguimento de boas práticas quando você está desenvolvendo?**

   Acho que levo muito em consideração seguir as boas práticas. Eu procuro sempre seguir aquilo que a gente aprendeu com as práticas e aquilo que foi acordado com o time em relação ao que são as boas práticas de cada time, porque pode variar um pouco. Mas com certeza eu levo no meu dia a dia esta questão bem a sério. Eu procuro fazer o meu código o mais *readable* possível, ou seja, fácil de se entender, que tenha significado. Eu procuro comentar bastante o meu código, quando possível, para que as pessoas, tanto outras pessoas que vão ler esse código, quanto eu mesmo daqui certo tempo posso pegar e ter necessidade de revisitar aquele código e implementar alguma coisa nova, ou reescrever ele de alguma forma, então é sempre importante seguir essas boas práticas, tanto a maneira que a gente escreve o código quanto comentários, etc. Para que outras pessoas, que possivelmente vão trabalhar nesse código no futuro, possam entender ele da melhor forma possível.

4. **Se você se depara com um código que não está bem estruturado, você o modifica? Quando sim e quando não?**

   Na grande maioria das vezes sim, modifico ele, tento estruturar ele da melhor forma possível se eu identificar que não está bem estruturado. Mas, por exemplo, quando a gente vê que tem outras coisas a serem feitas, como uma aplicação se planeja fazer um grande *refactor* por ter vários problemas de estrutura de código, várias boas práticas que não foram seguidas, etc. Então, levando em conta que se tem planejado uma grande reestruturação, talvez se eu tiver fazendo uma tarefa e notar que o código que eu estou trabalhando não está bem feito, eu vou avaliar se vale a pena fazer essa melhoria nesse momento, ou talvez esperar. E, às vezes, também, a gente leva em conta a prioridade e a rapidez com que a gente precisa resolver um problema... Se é algo que precisamos resolver muito rápido, e a gente consegue resolver de uma maneira mais simples sem modificar muito, então talvez a gente

primeiro faça essa tarefa e mantenha o código do jeito que está, mesmo que não esteja bem estruturado, ao menos para resolver aquele problema, e aí deixa uma reestruturação para depois.

5. **Quais são os fatores que levariam você a não seguir boas práticas no uso de Vue.js?**

   Acho que dificilmente existiria algum fator que me levaria a não seguir essas boas práticas. Eu acho que só se eu estivesse fazendo um POC, algo assim, em que eu precisasse desenvolver um conceito e provar ele rapidamente e aí, talvez, escrever o código de uma maneira mais rápida sem se preocupar com detalhes, aí talvez eu não seguisse algumas. Mas acho que só assim, pois no caso de uma aplicação que vai pra produção e que vai ter mais pessoas mexendo nela, ela tem que seguir alguns padrões mínimos.