Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Uma proposta de arquitetura para Single-Page Applications

Daniel de Jesus Oliveira

Trabalho de Graduação

Recife
11 de dezembro de 2017

Universidade Federal de Pernambuco
Centro de Informática

Daniel de Jesus Oliveira

Uma proposta de arquitetura para Single-Page Applications

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da computação.

Orientador: *Kiev Santos da Gama*

Recife
11 de dezembro de 2017

Aos indivíduos que diariamente contribuem para que o conhecimento seja compartilhado e reutilizado por meio das comunidades de código aberto.

Agradecimentos

Um problema com agradecimentos é a dificuldade em expressar essa gratidão de forma apropriada dadas as limitações de nossa linguagem. Dada sua linearidade, parece impossível não deixar parecer que se está dando mais importância a uma pessoa ou grupo em detrimento de outra apenas pela ordem de suas menções. E dada as limitações de espaço e da memória humana, é fácil deixar de citar certos elementos e dar assim a entender que estes não têm importância. Sendo assim, começo a seção deixando claro que agradeço profunda e igualmente a todos os que me ajudaram na minha jornada até aqui. Vocês sabem quem são.Dito isso, ainda gostaria de mencionar alguns.

Agradeço a meus pais por me suportar integralmente na minha jornada a Recife.

Agradeço a Kiev, meu orientador, pela atenção e auxílio durante a concepção e redação deste trabalho.

Agradeço a todos os meus amigos que me ouviram, opinaram e ajudaram ao longo dos estudos feitos aqui.

Agradeço a todos os professores que conseguiram me fazer amar ainda mais o desenvolvimento de software: em especial, Marcelo Linder, Fernando Castor, Augusto Sampaio.

Agradeço, finalmente, a Diana, que me confortou nos momentos de maior dificuldade e me ajudou a continuar focado no que eu realmente queria.

Part of my joy in learning is that it puts me in a position to teach; nothing, however outstanding and however helpful, will ever give me any pleasure if the knowledge is for my benefit alone.

—SENECA (Letters from a Stoic)

Resumo

Desde sua concepção, a web tem passado por inúmeras transformações. Inicialmente uma plataforma centrada em documentos, hoje a World Wide Web é também uma plataforma de aplicações ricas e complexas que atualizam o conteúdo de suas interfaces sem a necessidade de recarregar a página por completo, sendo chamadas de *single-page applications*. Com essa evolução, surgiu uma comunidade tão caótica quanto a própria humanidade; cada aspecto do desenvolvimento de uma aplicação web moderna encontra um vasto leque de soluções, e a escolha de uma delas pode ser estressante. Além disso, ocasionalmente pode haver a necessidade de realizar a troca de uma dessas soluções por outra, seja por mudança de requisitos ou por fatores externos como a licença de código aberto de alguma das soluções em uso. Nesse contexto, é crucial que essas aplicações apresentem uma arquitetura desacoplada, que permita a alteração de detalhes de implementação sem que outras partes do sistema sejam afetadas. Nesse trabalho, tentaremos identificar padrões que facilitem a criação de uma aplicação web com uma arquitetura que permita esse tipo de mudança.

Palavras-chave: JavaScript, Arquitetura de software, Arquitetura de aplicações web

Abstract

Since its conception, the web has been going through numerous transformations. Initially a document-centric platform, the World Wide Web today is also a platform for rich and complex applications that refresh their interfaces' content without the need for a full page reload, thus being called single-page applications. This evolution, among other factors, gave birth to a community as chaotic as mankind itself; each aspect of the development of a modern web application finds a wide array of solutions, and choosing between one of them can be stressful. Furthermore, ocasionally it may be desirable to change one of these solutions for another, be it due to a change in requirements or to external factors such as the open source license of some of the solutions being used. In this scenario it's essential that these applications present a loosely coupled architecture, allowing such implementation details to be altered without affecting more pieces of the system than necessary. In this work, we will try to identify patterns that facilitates the creation of a web application with such architecture.

Keywords: JavaScript, Software architecture, Web application architecture

Sumário

1 Introdução **1**

1.1 A evolução do JavaScript 2

1.1.1 A comunidade de código aberto JavaScript 3

1.2 Objetivo do trabalho 4

1.3 Estrutura do documento 4

1.4 Sumário do Capítulo 5

2 Aplicações de página única **6**

2.1 A anatomia de uma SPA 6

2.1.1 A criação da interface 6

2.1.1.1 Ligação de duas vias 7

2.1.1.2 Ligação de via única 8

2.1.2 Gerenciamento de estado 8

2.1.3 Geração do arquivo final 9

2.2 Desafios no desenvolvimento de SPAs 10

2.3 Sumário do Capítulo 10

3 Arquitetura de software **11**

3.1 Motivação 11

3.1.1 Decisões sobre detalhes de implementação 12

3.1.2 Adaptabilidade 12

3.1.3 Expansibilidade 13

3.2 Princípios 13

3.2.1 Princípio da Responsabilidade Única 14

3.2.2 Princípio Aberto/Fechado 15

3.2.3 Princípio da Inversão de Dependências 16

3.2.4 Arquitetura Limpa 17

3.3 Padrões comuns no desenvolvimento de SPAs 18

3.3.1 Componentes Contâiner 19

3.4 Desafios na aplicação de modelos de arquitetura em SPAs 20
3.5 Sumário do Capítulo 21

4 Proposta 22
4.1 Problema 22
4.2 Solução 22
4.3 Exemplo 23
4.3.1 Primeira iteração 24
4.3.2 Segunda iteração 25
4.4 Sumário do Capítulo 26

5 Análise 27
5.1 Sumário do Capítulo 29

6 Conclusão e trabalhos futuros 30

Lista de Figuras

2.1 Esquema do funcionamento da ligação de duas vias 7
2.2 Esquema do funcionamento da ligação de duas vias 8
2.3 Esquema do padrão de arquitetura proposta pelo Flux 8

3.1 Ilustração dos componentes do sistema exemplo 12
3.2 Arquitetura de uma aplicação multiplataforma com código compartilhado 14
3.3 Classe ControladorRegistroDespesa violando o Princípio da Responsabilidade Única 15
3.4 Classe ControladorRegistroDespesa refatorada para atender ao Princípio da Responsabilidade Única 15
3.5 Aplicação do Princípio de Inversão de Dependências 16
3.6 Divisão em camadas de uma aplicação como sugerido pela Arquitetura Limpa 17

4.1 Visão geral das dependências entre os elementos que compõem a primeira iteração da aplicação 24
4.2 Visão geral das dependências entre os elementos que compõem a segunda iteração da aplicação 26

5.1 Elementos afetados pelo teste na primeira iteração 28
5.2 Elementos afetados pelo teste na segunda iteração 29

Lista de Tabelas

5.1 Total de linhas de código do projeto em cada iteração antes da realização dos testes. 27

5.2 Resultados do cenário de troca da biblioteca Redux pela biblioteca Mobx nas duas versões do sistema em termos de arquivos alterados. 28

5.3 Resultados do cenário de troca da biblioteca Redux pela biblioteca Mobx nas duas versões do sistema em termos de linhas de código. 28

Capítulo 1
Introdução

Ao longo dos anos, a evolução das tecnologias e da própria sociedade fazem com que a forma de interagir com aplicações mude. Nesse contexto, o desenvolvimento de aplicações Web tem mudado bastante, com aplicações que realizam cada vez mais tarefas dentro do próprio navegador de seus usuários. Essas aplicações naturalmente impõem uma complexidade maior no processo de desenvolvimento, diferindo da natureza quase improvisada que se observava no uso de JavaScript em sistemas Web.

Atualmente, o uso de JavaScript se tornou comum entre as mais diversas plataformas. A popularidade da linguagem se torna clara através de pesquisas como a *Stack Overflow Developer Survey 2017* [1], realizada pela conhecida comunidade de desenvolvedores. Dentre mais de 64000 de seus usuários, 72.6% se denominam desenvolvedores Web. Destes, 81.7% utilizam JavaScript em seu trabalho. Além de aplicações que são executadas no navegador de seus usuários, hoje existem ferramentas que possibilitam a implementação de aplicações Desktop [2], aplicações nativas para smartphones [3], servidores [4] e até mesmo controle de robôs e dispositivos [5].

A comunidade de código aberto JavaScript é grande e ativa, e muitos dos problemas relacionados ao desenvolvimento de aplicações Web modernas encontram soluções prontas para reuso. O gerenciador de pacotes mais popular para a linguagem, o npm [6], possui quase meio milhão de pacotes registrados. No entanto, como em qualquer outro sistema de software, deve-se ter cautela ao decidir pelo uso de tais ferramentas. A arquitetura do sistema deve se manter flexível o bastante para tornar o código da aplicação estável à medida que mudanças ocorrem, visto que a maior parte do orçamento de um projeto de software é representada pela sua manutenção e evolução [7].

O repertório de trabalhos existentes que estudem abordagens arquiteturais no contexto de aplicações web do lado do cliente é escasso. Em [8], é feita uma análise do impacto de diferentes padrões de fluxo de dados na mantenabilidade de aplicações web. Apesar de apresentar resultados inconclusivos, a pesquisa apontou resultados levemente melhores dentre as bibliotecas que fazem uso de fluxo de dados unidirecional, que se tornou o padrão mais popular atualmente. Um estudo acerca da arquitetura de aplicações web é feito em [9], mas o trabalho foca apenas no lado do servidor dessas aplicações.

1.1 A evolução do JavaScript

Inicialmente concebida como um meio de registro e compartilhamento de documentos acadêmicos [10], a World Wide Web mudou muito desde seu nascimento em meados de 1990. À medida que foi se tornando mais acessível à população, a Web foi assumindo um crescente papel econômico, social e político na sociedade. A complexidade do conteúdo servido também aumentou, e páginas que se resumiam a documentos estáticos hoje se apresentam também como aplicações que se comportam similarmente a aplicações nativas. Essa mudança na complexidade do conteúdo se deve a dois fatores em especial: o surgimento do JavaScript em 1995, e dez anos depois, a popularização do Ajax [11].

A partir da necessidade de permitir uma maior interatividade em páginas web, em 1995 Brendan Eich criou uma linguagem de programação em apenas dez dias. Inicialmente chamada Mocha, em Dezembro do mesmo ano ela recebeu o nome JavaScript, numa tentativa de acelerar a popularização da linguagem após o recebimento de uma licença de marca da Sun Microsystems [11]. No ano seguinte, houve uma tentativa de padronização junto à Ecma International para que outros fabricantes de navegadores pudessem ter suas próprias linguagens de scripting, mas ainda oferecendo um conjunto base de funcionalidades sob as mesmas APIs. A esse padrão se deu o nome de ECMAScript (ou ES).

No entanto, a evolução do JavaScript não seguiu um caminho simples; Ao passo em que os primeiros navegadores competiam para se consolidar entre uma grande base de usuários, haviam divergências entre as APIs oferecidas pelos competidores para as implementações (nem sempre completas) de ES que ofereciam. Essas falhas motivaram a comunidade ao redor do desenvolvimento JavaScript a implementar polyfills, códigos que objetivam replicar APIs ausentes em certos navegadores utilizando JavaScript [12], e bibliotecas com foco em prover funcionalidade uniforme através de diversos navegadores, como jQuery [13].

Em 2005, Jesse James Garrett publicou um artigo técnico [14] que visava a popularização

de uma mudança no cenário do que era possível fazer com JavaScript: o artigo propôs uma abordagem na construção de aplicações web chamada Ajax. O nome é uma abreviação de *Asynchronous JavaScript + XML* (JavaScript assíncrono + XML), e apresentava uma mudança significativa no modelo tradicional de aplicações web: sob o modelo tradicional, as interações do usuário disparavam uma requisição HTTP para um servidor, que processava os dados necessários e servia uma nova página HTML como resposta à requisição. Com Ajax, a comunicação entre cliente e servidor se torna assíncrona através de uma camada adicional chamada motor Ajax, que solicita dados ao servidor e que realiza no lado do cliente todo o processamento que possa ser feito sem a necessidade de enviar ou solicitar dados ao servidor.

Com o uso de Ajax, um tipo de aplicação web se tornou amplamente popular: páginas que atualizam sua interface à medida que seus usuários interagem, sem a necessidade de recarregar todo o conteúdo. Conhecidas popularmente como *single-page applications* (SPAs), essas aplicações foram ganhando espaço por prover uma experiência mais próxima de aplicações nativas e por oferecer uma economia no uso da rede, visto que a comunicação cliente-servidor agora envolve majoritariamente dados, e não uma interface completa.

O problema dos padrões entre navegadores começou a ser solucionado em 2008. Num encontro entre membros do TC39, o comitê responsável pela manutenção dos padrões do ECMAScript, e representantes dos maiores fornecedores de navegadores web, foi feito um esforço para a criação de um padrão que contornasse as divergências existentes em um plano de evolução da linguagem [15]. A esse projeto foi dado o nome ECMAScript Harmony.

Em 2009, o conjunto do que é possível fazer com JavaScript começou a se tornar ainda maior. Num evento chamado JSConf EU, Ryan Dahl fez uma palestra [16][17] que mostrou um projeto chamado Node.js [4] que se baseava no motor de JavaScript V8 para permitir a execução de JavaScript no lado do servidor. No começo do ano seguinte, foi lançado junto ao Node.js um gerenciador de pacotes, o npm [6], que largamente facilitou a criação e o compartilhamento de bibliotecas de terceiros.

1.1.1 A comunidade de código aberto JavaScript

A comunidade de JavaScript se mostrou ativa o bastante para identificar as limitações que se apresentavam nas suas ferramentas e construir outras que corrigissem ou amenizassem tais falhas. Nesse ambiente em que a criação de ferramentas pelos próprios usuários delas é tão comum, temos uma situação semelhante à que tínhamos antes do ES Harmony.

Tomando como exemplo a tarefa de construção da interface de usuário, atualmente temos uma grande quantidade de opções de bibliotecas e frameworks para facilitar o desenvolvimento:

uma rápida pesquisa levanta 7 alternativas: Angular [18], Ember [19], React.js [20], Vue.js [21] são algumas das alternativas escritas em JavaScript. Temos ainda mais algumas baseadas em outras linguagens como Om [22] e Reagent [23], baseadas em ClojureScript [24], e Elm [25], que é uma linguagem distinta compilada para JavaScript. O mesmo cenário se repete em outras tarefas relevantes ao desenvolvimento de aplicações web, como o gerenciamento de estado da aplicação.

A comunidade também encontrou outra solução para o ainda existente problema de inconsistência entre navegadores: compilar o código-fonte de modo a simular certas funcionalidades da linguagem ainda não presente em todos os navegadores em função de APIs já suportadas por todos os navegadores. Compiladores como o Babel [26] são utilizados em larga escala atualmente, atingindo milhões de downloads mensalmente no último ano [27].

1.2 Objetivo do trabalho

Levando em conta o cenário de constante mudança nas ferramentas utilizadas na comunidade de desenvolvimento web, objetiva-se neste trabalho amenizar os custos da mudança de tais ferramentas. Espera-se por meio deste trabalho chegar a um padrão por meio de que se pode realizar a troca de uma dessas ferramentas por outra com um mínimo de impacto na mantenabilidade do resto do projeto. A busca por esse padrão foi feita com o auxílio de princípios e modelos que visam guiar o projeto da arquitetura de um sistema de software, bem como sua implementação e manutenção.

1.3 Estrutura do documento

No Capítulo 2, será feita uma análise mais aprofundada dos elementos que formam uma aplicação web moderna, bem como dos desafios relacionados ao desenvolvimento de uma. Então, no Capítulo 3 serão apresentados a motivação e os princípios por trás da disciplina de arquitetura de software, bem como padrões do estado da prática no desenvolvimento de SPAs. Finalmente, será explanado no Capítulo 4 um conjunto de padrões que visam prover uma estrutura estável para aplicações web, bem como o processo utilizado para chegar a esse conjunto. Esses padrões serão analisados no Capítulo 5, de modo a averiguar sua eficácia no escopo proposto.

1.4 Sumário do Capítulo

Neste Capítulo, foi possível entender a evolução pela qual JavaScript passou até o cenário atual, bem como alguns problemas gerados pelo crescimento de sua comunidade. Além disso, foi apresentado o conceito de SPAs, cuja popularidade e presença na rede mundial de computadores não pode ser ignorada. No próximo capítulo, esse tipo de aplicação será examinado mais detalhadamente, levando em conta a composição de tais sistemas e os desafios encontrados ao desenvolvê-los.

Capítulo 2
Aplicações de página única

O processo de convergência do ECMAScript facilitou o desenvolvimento do tipo de aplicação web apresentado em [14]: aplicações que eram de fato compostas por uma única página, cujo conteúdo era substituído à medida que processava as interações dos usuários. Por se assemelhar a aplicações nativas, essas aplicações ganharam popularidade. A esse tipo de aplicação se dá o nome Aplicação de Página Única, conhecido popularmente como SPA.

2.1 A anatomia de uma SPA

Como em qualquer outra aplicação web, o ponto de entrada de uma SPA é um arquivo de marcação, escrito em *HyperText Markup Language*, ou HTML. A diferença está no conteúdo desse arquivo: geralmente, não há nenhum elemento de interface definido, apenas metadados e a importação de código JavaScript. Nesse arquivo JavaScript é onde toda a aplicação está contida: geração de elementos de interface, lógica de negócios e comunicação com outros serviços.

De modo a entender aplicações desse tipo mais profundamente, vamos analisar o estado da prática de alguns dos elementos que compõem SPAs. A complexidade desse tipo de aplicação pode variar amplamente, e sendo assim iremos nos concentrar em quatro aspectos: a interface de usuário, o gerenciamento de estado e a geração do código final.

2.1.1 A criação da interface

Parte do código de uma SPA é responsável por manter a interface em sincronia com os dados da aplicação. Bibliotecas e frameworks como as citadas na seção 1.1.1 se utilizam de alguma forma de linguagem auxiliar para gerar templates que representam os elementos HTML que formarão o conteúdo da página. Dessa forma, é facilitada a componentização dos elementos de interface. Comumente, os componentes são definidos declarativamente, e a apresentação e atualização dos elementos de interface que os compõem é resolvida pelo funcionamento interno da ferramenta de interface utilizada.

Para manter a representação dos elementos sincronizada na interface, são feitas alterações no *Document Object Model* (DOM), a estrutura usada pelos navegadores para os elementos HTML em exibição na página. Operações de manipulação do DOM costumam ser de custo computacional elevado, especialmente quando envolvem uma quantidade elevada de elementos manipulados. Isso se dá porque tais manipulações podem acarretar a necessidade de recalcular grande parte do layout da página. Dessa forma, apenas substituir o DOM por inteiro se torna inviável.

Para contornar esse custo, normalmente se utiliza uma representação auxiliar do DOM, que não tem relação direta com a interface sendo exibida e fica contida apenas em memória. A partir dessa estrutura, computa-se o conjunto mínimo de alterações que devem ocorrer no DOM quando ocorre alguma alteração no estado da aplicação. Essa abordagem é utilizada em praticamente todas as soluções de construção de interfaces Web atualmente. O React.js, por exemplo, faz uso de um algoritmo de diferenciação de árvores com o auxílio de heurísticas [28] para descobrir o que deve ser alterado na página.

Outro aspecto importante na geração da interface de SPAs é a maneira como é feita a ligação entre os dados da aplicação e a interface que apresenta esses dados. As abordagens usadas pelas ferramentas no estado da prática se dividem em duas categorias: ligações de duas vias e ligações de via única.

2.1.1.1 Ligação de duas vias

Na ligação de duas vias, os elementos de interface e o estado da aplicação estão fortemente ligados. Através de entidades que observam mudanças tanto na interface quanto no estado da aplicação, alterações feitas nestes elementos implicam uma alteração no próprio estado. Esses observadores geralmente são criados internamente pelos frameworks que os utilizam. Exemplos de ferramentas que oferecem ligação de duas vias são a biblioteca Vue.js e o framework AngularJS. O modelo é ilustrado na Figura 2.1.

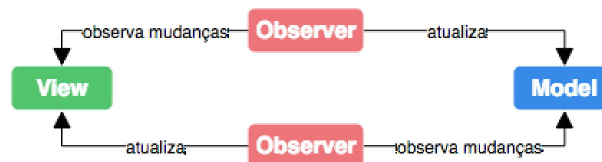

Figura 2.1 Esquema do funcionamento da ligação de duas vias. Adaptado de [29].

2.1.1.2 Ligação de via única

Na ligação de via única, os elementos de interface não são observados por mudanças. Assim, o conteúdo da interface toma um aspecto puramente de apresentação, e as alterações de estado são feitas explícitas através do tratamento de eventos disparados pela interação do usuário com a interface da aplicação. Atualmente essa abordagem tem sido favorecida por tornar o comportamento da aplicaçao mais previsível. A biblioteca React.js só oferece a ligação de via única para a apresentação dos dados da interface. O funcionamento da ligação de via única é ilustrado na Figura 2.2.

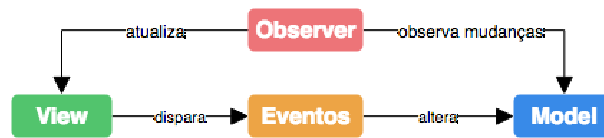

Figura 2.2 Esquema do funcionamento da ligação de duas vias. Adaptado de [29].

2.1.2 Gerenciamento de estado

O gerenciamento de uma aplicação pode ser uma tarefa complexa, com muitos elementos do sistema interagindo com o estado da aplicação. Com o intuito de uniformizar o acesso e as mutações do estado de suas aplicações, o Facebook propôs o padrão arquitetural Flux, que apresenta um fluxo de dados unidirecional. No Flux, o estado só pode ser modificado através do disparo de objetos que representam ações realizadas dentro do sistema, como descrito na Figura 2.3.

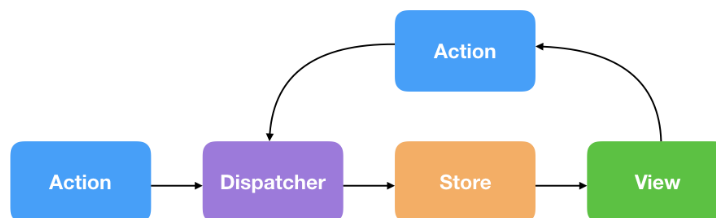

Figura 2.3 Esquema do padrão de arquitetura proposta pelo Flux. Ações são disparadas através do *Dispatcher*, que as repassa aos objetos que contém partes do estado da aplicação, as *Stores*. Outras partes da aplicação podem então se tornar observadores desse estado, sendo notificadas quando ele sofre alterações, e disparando novas ações em reação a interações do usuário. Adaptado de [30].

Existem ferramentas que oferecem outras abordagens para gerenciar o estado de uma aplicação, mas em geral o estado da prática faz uso do conceito de fluxo de dados unidirecional. Por exemplo, o MobX faz uso do padrão Observer para detectar mudanças no objeto que contém o estado. Existem também soluções que são adaptações do Flux, como o Redux [31]. Inspirado por conceitos de programação funcional [32], a biblioteca conseguiu simplificar conceitos do Flux, como a existência de múltiplas *Stores* e a implementação de alterações de estado por meio de funções puras.

Uma nota importante sobre o Redux é o conceito de middleware [33]. No contexto da biblioteca, middlewares são uma implementação do padrão *Decorator* [34] sobre o dispatcher mostrado na Figura 2.3. Middleware interceptam ações despachadas e podem ter objetivos variados, geralmente utilizados para habilitar o despacho de outros tipos de ações, ferramentas auxiliares no desenvolvimento e na execução de tarefas assíncronas.

Com o uso de soluções de gerenciamento de estado, também é possível atingir outro resultado interessante: o código responsável por definir a interface se torna desprovido de estado interno, tomando um caráter funcional: os elementos se tornam funções puras do estado da aplicação, desprovidos de qualquer estado interno.

2.1.3 Geração do arquivo final

No desenvolvimento de uma aplicação web, cada arquivo JavaScript deve ser incluso na página HTML servida através de uma tag script. Assim, se torna necessário incluir uma tag para cada arquivo fonte do projeto ou concatenar o conteúdo de todos em um só arquivo, e ambos os processos são problemáticos. A inclusão das tags na página HTML é altamente propensa a esquecimentos e enganos por parte dos desenvolvedores. Além disso, as duas abordagens podem se tornar complicadas caso a ordem de inclusão dos arquivos seja relevante.

O estado da prática consiste no uso de ferramentas como o Webpack [35], que analisam as dependências do código-fonte do projeto e geram um só arquivo que será servido pelo navegador. Cada arquivo é redigido como um módulo JavaScript, importando suas dependências como em muitas outras linguagens e frameworks. Além disso, tais ferramentas são capazes de realizar eliminação de código morto no arquivo resultante, diminuindo o tamanho total.

2.2 Desafios no desenvolvimento de SPAs

Essa complexidade adicional no lado do cliente torna inviável a criação de aplicações em escala utilizando apenas as APIs disponibilizadas pelos browsers, que são de baixo nível e ainda não completamente uniformes até o presente momento. Assim, ferramentas de código aberto como as citadas na subseção 1.1.1 se tornam essenciais para abstrair tarefas repetitivas ou complexas.

O desenvolvimento de uma SPA pode se tornar um processo complicado rapidamente. Considerando o problema de gerenciamento de estado da aplicação, a escolha pode não ser tão simples mesmo com soluções consolidadas na área. O Redux, por exemplo, pode não ser tão apropriado em situações em que a aplicação muda com uma frequência elevada, cenário frequente entre startups em fase inicial. À medida que a complexidade dessas aplicações aumenta, pode ser interessante dispor de certas funcionalidades oferecidas pela biblioteca. A troca de uma alternativa mais apropriada durante a fase de implementação por outra mais robusta nem sempre é simples.

A escolha de ferramentas também pode ser afetada por fatores externos. Por exemplo, esse ano houve uma grande discussão na comunidade sobre a validade do uso de algumas ferramentas criadas pelo Facebook e disponibilizadas sob a licença *Facebook BSD+Patents*. Entre essas ferramentas estava o React.js, biblioteca de front-end que recebe centenas de milhares de downloads diariamente [36]. O problema surgiu quando a Apache Foundation [37] baniu o uso de quaisquer ferramentas liberadas sob a licença [38]. Posteriormente, o Facebook mudou vários de seus produtos para a licença MIT, aceita pela Apache Foundation. O problema se resolveu, mas poderia ter causado a migração de vários produtos já existentes para outras bibliotecas de front-end para evitar problemas.

2.3 Sumário do Capítulo

Neste Capítulo, foi possível obter um entendimento maior sobre SPAs e como funcionam. Também foram apresentados alguns dos desafios encontrados ao desenvolver tais sistemas, diante da complexidade que eles podem adotar. No próximo Capítulo, serão apresentados alguns conceitos de Arquitetura de Software que serão de grande utilidade ao contornar alguns dos desafios apresentados nesse Capítulo.

CAPÍTULO 3

Arquitetura de software

A arquitetura de software é a disciplina que orienta, entre outras coisas, a relação entre os elementos de um sistema. É importante deixar claro que o termo “elementos” não se refere apenas a classes e objetos, principalmente se levando em conta que o desenvolvimento de SPAs mescla características de diferentes paradigmas de linguagens de programação.

Segundo [39], os elementos de um sistema se classificam em três categorias: dados, processamento e conexão. Os elementos de dados são transformados pelos de processamento, e a saída dos elementos de processamento são ligadas à entrada de outros através dos elementos de conexão. Nesse Capítulo entenderemos melhor esses e alguns outros aspectos sobre a área de arquitetura de software e as particularidades de sua aplicação no desenvolvimento de SPAs.

3.1 Motivação

Existem estudos [40] acerca das dificuldades de realizar mudanças num sistema de software ao longo de sua manutenção e evolução, num processo traduzido livremente como *degradação de código*. Entre os motivos que podem causar essa degradação, estão um modelo de arquitetura inadequado para a aplicação, e a violação dos princípios do projeto. Assim, se torna relevante para o time responsável pelo projeto que sua arquitetura seja definida de forma que seja adequada para o sistema em questão, e que seus princípios sejam seguidos por toda a equipe de desenvolvimento [41].

Outro fruto da definição e aplicação desses princípios é o desacoplamento entre os elementos relacionados às regras de negócio e os dispositivos de entrada e saída. Tais dispositivos se tornam meramente detalhes de implementação, e o ato de desacoplá-los traz algumas vantagens significativas ao longo do ciclo de vida do sistema. Serão observadas três dessas vantagens nas próximas sessões: a qualidade da tomada de decisões sobre o sistema, a adaptabilidade do sistema e sua expansibilidade.

3.1.1 Decisões sobre detalhes de implementação

Como fala Robert Martin em [42], uma das funções do arquiteto de software é "deixar o máximo de opções abertas pelo maior tempo possível". Em outras palavras, o papel do arquiteto é projetar sistemas de forma que seus elementos ofereçam flexibilidade quanto a suas dependências.

Considere, por exemplo, uma aplicação de gerenciamento de finanças pessoais. Cada nova entrada de um gasto ou receita modifica uma agregação de dados usada na exibição de um relatório para o usuário. O elemento responsável pela atualização dessa agregação não precisa saber se os dados da nova transação foram inseridos manualmente pelo usuário ou através de uma API que acessa os dados bancários do usuário diretamente. Isto é, não há a necessidade de que haja código dentro desse elemento que interaja diretamente com o dispositivo de entrada dos dados ou que faça requisições a essa API. Analogamente, esse elemento não precisa saber se o sistema está armazenando seus resultados em um banco de dados ou em arquivos de texto. Os elementos do sistema não dependem do meio através do qual os dados de que precisam são obtidos. Essa relação é ilustrada pela Figura 3.1.

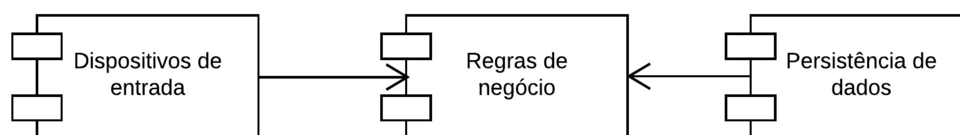

Figura 3.1 Ilustração dos componentes do sistema exemplo. As setas indicam as dependências entre os componentes. Note como os dispositivos de entrada e o componente responsável pela persistência de dados dependem das regras de negócio, e não o contrário. Entenderemos como isso acontece na seção 3.2.

Esse desacoplamento permite que regras de negócio sejam completamente independentes dos detalhes de implementação. Dessa forma, a decisão sobre aspectos como os citados no Capítulo 2 podem ser adiadas até que se tenha informação suficiente para tomá-las de forma mais consciente, possivelmente com uma chance menor de que seja necessário voltar atrás em tais decisões.

3.1.2 Adaptabilidade

Mesmo quando se tem acesso a uma quantidade maior de informação na tomada de decisão sobre detalhes de implementação, mudanças podem ser necessárias. Soluções existentes podem

se tornar obsoletas, e os requisitos de uma aplicação podem mudar de forma que tornem uma decisão anterior inviável para a continuidade do projeto.

Reutilizando o exemplo anterior, pode-se imaginar um cenário em que o banco de dados utilizado pela aplicação não supre os requisitos de performance para a carga de dados que produz. Assim, se faz necessária a análise e substituição por uma alternativa, bem como alterações no código responsável pela comunicação com o banco de dados que era utilizado.

Nesse contexto, se torna igualmente desejável que os elementos do sistema se relacionem de forma que saibam o mínimo necessário sobre outras partes do sistema. Por meio de princípios como o da Inversão de Dependências [43], visto em detalhes na seção 3.2, pode-se trocar certas partes do sistema sem que seja necessário alterar o código de outros elementos.

3.1.3 Expansibilidade

Do ponto de vista de negócios, pode ser interessante que um sistema esteja disponível para o usuário em múltiplas plataformas com que ele interaja. No entanto, esse processo pode se tornar custoso à medida que essa expansão para outras plataformas cria a necessidade de manter múltiplas bases de código.

Atualmente, esse custo pode ser mitigado através de soluções que permitem escrever aplicações para diferentes plataformas na mesma linguagem de programação. Algumas dessas soluções foram citadas na introdução desse trabalho: com JavaScript, é possível escrever aplicações desktop com Electron e para smartphones com React Native.

Para aproveitar ao máximo essas soluções, se torna importante que as regras de negócio sejam independentes dos detalhes de implementação. Assim, é possível que se tenha uma aplicação servida para várias plataformas que compartilham uma grande parte de suas bases de código entre si, como ilustrado na Figura 3.2.

3.2 Princípios

A arquitetura de um sistema de software se baseia em um conjunto de regras. Essas regras são repersentadas por padrões e princípios que definem, classificam e restringem elementos do sistema e como eles se relacionam.

A definição dessas regras é relevante ao longo de todo o ciclo de vida do sistema. Durante a fase de análise e projeto, elas ajudam a dar forma ao sistema a ser implementado. Durante o desenvolvimento e a manutenção, esses padrões oferecem uma abordagem uniforme ao se

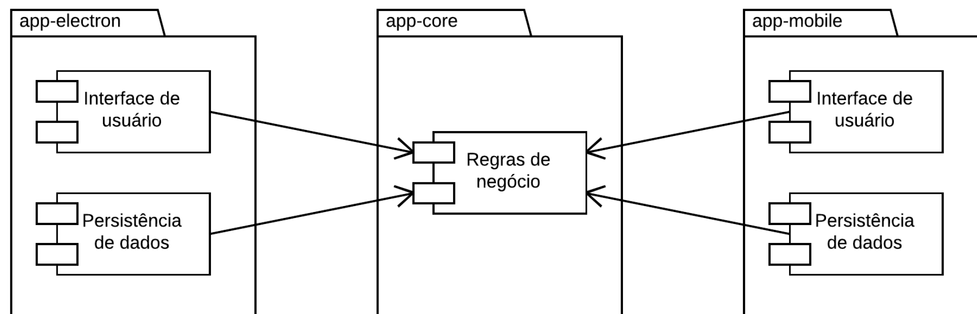

Figura 3.2 Arquitetura de uma aplicação multiplataforma com código compartilhado. A implementação das regras de negócio do produto é compartilhada entre o código das duas aplicações.

atacar certos problemas que surgem naturalmente à medida que o sistema sofre mudanças. O trabalho feito em [39] atribui em partes o surgimento desses problemas à *erosão arquitetural* ou *deriva arquitetural* [40], também chamada *degeneração arquitetural* em outros trabalhos [44]. O efeito consiste na violação das regras da arquitetura definida para a aplicação, culminando na mudança gradual da arquitetura do sistema para uma diferente da que fora inicialmente planejada.

Será dado foco a alguns dos princípios de projeto mostrados em [45], e no estilo arquitetural chamado *Arquitetura Limpa*, apresentado pioneiramente em [46] e *a posteriori* no livro do mesmo autor [42].

3.2.1 Princípio da Responsabilidade Única

Descrito inicialmente em [47] e [48], o Princípio da Responsabilidade Única dita que um elemento deve possuir apenas um motivo para mudar. Isto é, deve-se restringir as responsabilidades de cada elemento do sistema de modo que ao passo que ele muda, apenas um conjunto bem definido de elementos precisará ser modificado. Se um elemento carrega em si mais de uma responsabilidade, mudanças relacionadas a uma delas pode afetar o funcionamento de outras das suas funções.

Reutilizando o exemplo anterior da aplicação de finanças pessoais, pode-se considerar uma classe ControladorRegistroDespesa, responsável pela execução do caso de uso de registro de despesas. A classe está representada na Figura 3.3.

A classe ControladorRegistroDespesa está violando o Princípio da Responsabilidade Única ao tratar das regras de negócio em geral e também persistir a despesa localmente na aplicação.

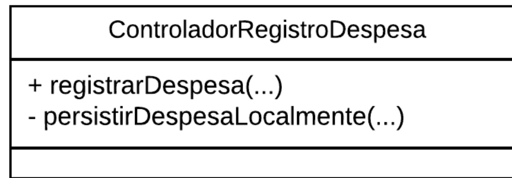

Figura 3.3 Classe ControladorRegistroDespesa violando o Princípio da Responsabilidade Única.

Isso pode dificultar a manutenção da classe à medida que sua implementação sofre mudanças. Além disso, caso suas responsabilidades distintas compartilhem código, um desenvolvedor desatento pode fazer com que a classe passe a apresentar comportamento incorreto. É possível melhorar a situação dividindo o nosso controlador em duas classes, como demonstrado na Figura 3.4.

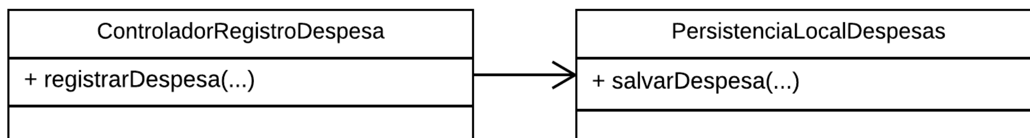

Figura 3.4 Classe ControladorRegistroDespesa refatorada para atender ao Princípio da Responsabilidade Única.

3.2.2 Princípio Aberto/Fechado

Criado em [49] por Bertrand Meyer, o Princípio Aberto/Fechado diz respeito à extensibilidade de um sistema. Ele afirma que os elementos de um sistema devem estar abertos para extensão, mas fechados para modificação. Em outras palavras, quando uma mudança no sistema acarreta uma série de mudanças em outras partes do sistema, o Princípio Aberto/Fechado aconselha a refatorar o sistema de modo que mudanças futuras do mesmo tipo possam ser alcançadas pela adição de código novo, deixando o código já existente intocado.

É simples ilustrar o princípio através de uma prática comum no desenvolvimento de SPAs: a ligação entre uma ferramenta de gerenciamento de estado e os componentes de interface da aplicação. De modo a simplificar a explicação, serão usados Redux e React.js como exemplo de tais ferramentas. Comumente, a ligação entre os dois é feita no arquivo fonte do próprio componente.

Isso torna o código relacionado à implementação dos componentes muito vulnerável a mudanças. Por exemplo, se as APIs do Redux mudarem após um upgrade da biblioteca, todos os

componentes ligados ao gerenciamento de estado deverão ter seus códigos modificados. Num cenário ainda pior, caso seja tomada uma decisão de migrar o gerenciamento de estado para outra ferramenta, todo o código responsável pela ligação entre o Redux e os componentes React deverá ser alterado, de forma muito similar em vários arquivos.

Isso poderia ser evitado, por exemplo, por meio de uso de interfaces que permitam que os componentes interajam com o sistema sem depender diretamente do Redux ou de qualquer outra ferramenta. Essa técnica e outras serão discutidas no Capítulo 4.

3.2.3 Princípio da Inversão de Dependências

O princípio da Inversão de Dependências postula que módulos de alto nível não devem depender de módulos de baixo nível diretamente [43]. Isto é, essas dependências devem ocorrer através de abstrações definidas pelas interfaces requeridas pelos módulos de alto nível. Esse princípio ocorre frequentemente na implementação de serviços, de modo a abstrair certos detalhes como a comunicação com a persistência de dados do sistema ou com APIs externas. Diante de mudanças, Se torna necessário apenas modificar a realização das abstrações de que dependem os módulos de alto nível.

O exemplo da Figura 3.4 viola o Princípio da Inversão de Dependências. O ControladorRegistroDespesa depende diretamente da classe PersistenciaLocalDespesas, e não há nenhum mecanismo que impeça o controlador de precisar de modificações quando a classe de persistência mudar. Uma solução é fazer com que o controlador defina uma interface através da qual se define os métodos de que precisa. Cabe, então, à classe de persistência realizar essa interface. Essa relação está ilustrada na Figura 3.5.

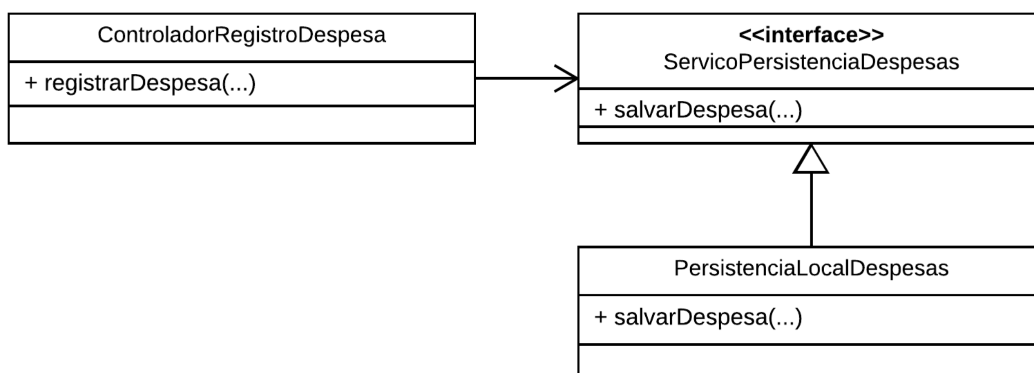

Figura 3.5 Aplicação do Princípio de Inversão de Dependências.

No contexto de desenvolvimento de SPAs, esse princípio pode levantar dúvidas: como é

possível estabelecer essa interface numa linguagem dinâmica como JavaScript? A resposta é simples: não é possível estabelecer essa interface explicitamente. Uma solução encontrada foi abstrair essa interface através de uma função responsável por instanciar a implementação concreta, expondo os métodos da interface através de um objeto simples (ou no jargão da linguagem, um *Plain Old JavaScript Object*, ou POJO).

3.2.4 Arquitetura Limpa

A Arquitetura Limpa foi um modelo de arquitetura proposto em 2012 por Robert C. Martin [46] e reapresentado em [42]. A abordagem proposta pelo modelo é similar a outras, como a Arquitetura Hexagonal [50, 51]. O conceito central é tratar as regras de negócio como o "núcleo" da aplicação. Detalhes de implementação como dispositivos de entrada/saída e componentes de interface gráfica se tornam apenas periféricos ao sistema, facilmente substituíveis.

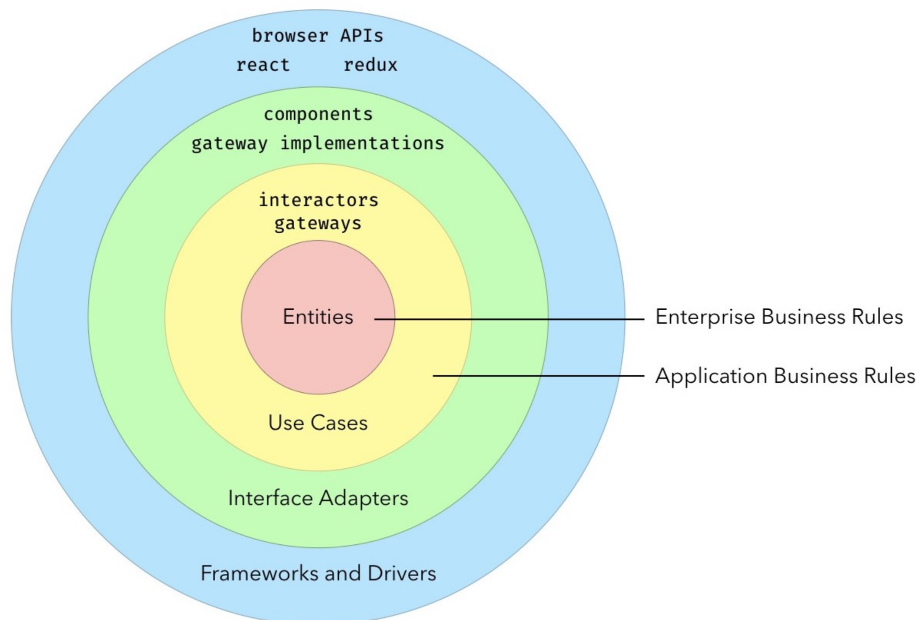

Figura 3.6 Divisão em camadas de uma aplicação como sugerido pela Arquitetura Limpa.

No conceito da Arquitetura Limpa, os elementos se dividem em camadas que se comunicam em confirmidade com um conjunto de regras. Em [46] são propostas quatro camadas, mas a quantidade e o significado de cada camada pode variar entre aplicações distintas. As camadas propostas estão ilustradas na figura 3.6, e são:

- **Entidades**

  Essa camada é a mais interna de todo o sistema. As entidades representam os elementos mais básicos de negócios de uma aplicação. No caso de uma base de código compartilhada entre múltiplas aplicações, as entidades compõem a maior parte do código compartilhado.

- **Casos de Uso**

  A camada de casos de uso representa os elementos responsáveis pela execução de cada um dos casos de uso de uma aplicação. Eles provém uma interface para a camada mais externa e agem como uma central que se comunica com outras partes do sistema.

- **Adaptadores de Interface**

  A camada de Adaptadores de Interface é a fronteira entre os Casos de Uso e as ferramentas que permitem que o sistema se comunique com o mundo externo: bancos de dados, interfaces gráficas e servidores Web. Como o nome da camada sugere, elementos pertencentes a ela agem como mediadores entre as camadas que os envolve.

- **Drivers e frameworks**

  A camada de Drivers e Frameworks contém as próprias ferramentas utilizadas pelo seu sistema. Normalmente não se escreve código nessa camada, que inclui bibliotecas como React e Redux.

Regendo as relações entre as camadas existentes numa aplicação, existe uma regra imposta pela Arquitetura Limpa:

> Um elemento de uma camada mais interna não pode depender de um elemento de uma camada mais externa.

Isto é, uma Entidade não pode depender de um Caso de Uso, da mesma forma que um Caso de Uso não pode depender de um Adaptador de Interface. O relacionamento de um elemento com outro pertencente a uma camada mais externa deve ocorrer através da Inversão de Dependências descrita na Seção 3.2.3.

3.3 Padrões comuns no desenvolvimento de SPAs

Existem práticas comuns na comunidade de desenvolvimento Web que visam atender a princípios arquiteturais bem estabelecidos como os citados na Seção 3.2. Esta seção visa explanar

algumas dessas práticas, que serão relevantes em seções posteriores. Neste trabalho, serão demonstrados apenas práticas relevantes ao uso da biblioteca de front-end React.js [20], devido a restrições de escopo.

3.3.1 Componentes Contâiner

Um componente React é comumente representado por uma classe que é uma especialização da classe `Component`, parte da biblioteca, e o método `render` do componente implementado define os elementos de interface a serem apresentados no navegador. Nos componentes também são implementados métodos para tratar eventos lançados através da interação do usuário com a aplicação, bem como os chamados *métodos de ciclo de vida* [52]. Estes são chamados automaticamente diante de acontecimentos como a primeira exibição do componente na interface da aplicação ou a atualização do estado interno de um componente. Um exemplo de componente que carrega de um servidor a cotação atual do bitcoin e a exibe na interface da aplicação é exibida no código abaixo.

```
class MessyComponent extends React.Component {
  state = {
    isLoading: true,
    btcValue: null
  }

  render() {
    return this.state.isLoading ? (
      <h1>Loading latest bitcoin value...</h1>
    ) : (
      <h1>The current bitcoin value is ${this.state.btcValue}</h1>
    )
  }

  componentDidMount() {
    fetch('https://www.example.com/btc')
      .then(response => response.json())
      .then(data => {
        this.setState({
          isLoading: false,
          btcValue: data.btcValue
        })
      })
  }
}
```

Listing 3.1 Componente React sem aplicação do padrão de Contâineres.

O componente acima possui múltiplas responsabilidades, violando o Princípio da Responsabilidade Única. Com o intuito de realizar a separação dessas responsabilidades, um padrão se estabeleceu e popularizou. O padrão consiste em dividir os componentes de uma aplicação em duas categorias:

1. **Componentes Contâiner** são responsáveis por se comunicar com serviços externos e por conter todo o estado relevante aos componentes utilizados por ele;

2. **Componentes de apresentação** são componentes cujo único propósito é definir os elementos de interface a serem exibidos

Um exemplo da mesma aplicação demonstrada acima depois da aplicação do padrão de Componentes Contâiner se encontra abaixo:

```
class PresentationalComponent extends React.Component {
  render() {
    return this.props.isLoading ? (
      <h1>Loading latest bitcoin value...</h1>
    ) : (
      <h1>The current bitcoin value is ${this.props.btcValue}</h1>
    )
  }
}

class ContainerComponent extends React.Component {
  state = {
    isLoading: true,
    btcValue: null
  }

  render() {
    return <PresentationalComponent
              isLoading={this.state.isLoading}
              btcValue={this.state.btcValue} />
  }

  componentDidMount() {
    fetch('https://www.example.com/btc')
      .then(response => response.json())
      .then(data => {
        this.setState({
          isLoading: false,
          btcValue: data.btcValue
        })
      })
  }
}
```

Listing 3.2 Componente React após a aplicação do padrão de Componentes Contâiner.

Com a aplicação do padrão de Contâineres, se torna possível separar completamente o código da interface do código responsável pelas regras de negócio do sistema. No entanto, essa abordagem ainda enfrenta problemas que serão apresentados no Capítulo 4.

3.4 Desafios na aplicação de modelos de arquitetura em SPAs

Em 2000, Pressman falou da relutância acerca da aplicação de princípios consolidados de engenharia em aplicações web [53]. Na época, SPAs ainda não haviam tomado a popularidade que têm hoje, mas o discurso dele ainda se aplica. Aplicações de página única são comumente desenvolvidas dentro de prazos curtíssimos, e muitas vezes, técnicas e métodos mais elaborados são tidos como exagero.

As bibliotecas e frameworks utilizados em SPAs são muitas vezes baseados em conceitos muito distintos. Assim, se torna difícil criar abstrações eficazes que não acabem complicando demais o desenvolvimento do produto. Além disso, a implementação dessas abstrações aumenta o tamanho do código servido ao navegador do cliente, resultando em carregamentos de página mais lentos [54].

No entanto, existem alguns padrões que podem ser aplicados de modo a obter melhoras significativas no projeto de uma SPA. No Capítulo seguinte, será apresentada uma proposta a ser aplicada no projeto de sistemas web de modo a atingir as vantagens elucidadas na seção 3.1.

3.5 Sumário do Capítulo

Nesse Capítulo, foi apresentada a importância da Arquitetura de Software e princípios que auxiliam no projeto de sistemas que evoluem de forma saudável e com um baixo custo. No próximo Capítulo, será feito uso dos princípios aqui apresentados de modo a propor um conjunto de padrões a serem aplicados num SPA de modo a tornar sua arquitetura mais resiliente a mudanças.

Capítulo 4
Proposta

Em [55], se atribui a origem da arquitetura de um sistema a três fontes: *roubo*, *método* e *intuição*. O roubo se refere ao ato de se inspirar em arquiteturas já existentes, imitando toda ou parte delas. O método diz respeito ao processo sistemático de determinar a arquitetura de um sistema a partir de seus requisitos e restrições, baseado em princípios e heurísticas bem definidas. A intuição fala do processo cognitivo por parte do arquiteto, que de suas experiências e pontos de vista pode trazer algo novo ao projeto.

Nesse trabalho, tentou-se usar ao máximo as duas primeiras fontes de modo a chegar a um padrão que satisfizesse aos critérios estabelecidos de qualidade. Neste Capítulo, explicaremos o problema a ser resolvido, os passos a serem seguidos de modo a aliviá-lo e um exemplo utilizado durante os estudos desse trabalho.

4.1 Problema

Dois problemas relacionados à manutenção de aplicações web foram considerados nesse trabalho:

1. O custo da mudança de dependências externas;
2. A alta concentração de responsabilidades em alguns componentes da aplicação.

Considerando o cenário de constante mudança no desenvolvimento Web descrito no Capítulo 1.1, o item 1 é de suma importância no sentido de proteger aplicações existentes em caso de obsolescência de soluções em uso no sistema desenvolvido.

4.2 Solução

A solução apresentada se inspira nos princípios apresentados no Capítulo 3 e num padrão de refatoração apresentado em [56], a divisão de classes que retém em si um excesso de respon-

sabilidades distintas, as chamadas *god classes* [57]. Os passos a serem seguidos consistem em:

1. Aplicar o Princípio da Inversão de dependências aos serviços externos utilizados, como a comunicação com o back-end e a realização de mutações no estado da aplicação;

2. Abstrair a ligação entre o gerenciamento de estado da aplicação e os componentes de interface através da Inversão de Dependências;

3. Quebrar elementos que retém responsabilidades não-relacionadas como sugerido em [56].

É importante notar que JavaScript não oferece o conceito de interfaces. A aplicação do Princípio da Inversão de Dependências nesse caso se dá por meio de funções que tem como função instanciar objetos com um determinado conjunto de propriedades e métodos, e a manutenção da integridade dessa interface fica por conta dos desenvolvedores. Com o intuito de permitir o uso de interfaces propriamente ditas, também pode-se fazer uso de ferramentas que permitam o desenvolvimento da aplicação com tipagem estática, como a linguagem TypeScript [58] ou o verificador de tipos Flow [59].

4.3 Exemplo

Nesta Seção, será descrita a arquitetura de uma aplicação desenvolvida para a realização de testes acerca da eficácia dos passos descritos na Seção 4.2. O desenvolvimento foi realizado em duas iterações, de modo a permitir a comparação dos resultados dos testes em cada cenário. Neste Capítulo, será dado foco ao estudo dos pontos fracos e fortes dos resultados de cada iteração.

Por meio de discussões com outros indivíduos com experiência de mercado e estudantes, concluiu-se também que seria desejável ter como ponto de partida um projeto construído sem esses princípios em mente. Essa decisão foi tomada de modo a facilitar a adesão da comunidade de desenvolvimento web, chegando a um resultado que não é composto exclusivamente por práticas incomuns no dia-a-dia dos desenvolvedores.

Foi realizada, então, a concepção e implementação de uma SPA que tivesse complexidade suficiente para ser de utilidade nos estudos, mas que fosse simples o bastante para se encaixar nas restrições de tempo e escopo do trabalho. A aplicação implementada é um sistema de salas de bate papo em que os usuários podem criar salas e participar de conversas em salas já

existentes. Para que o escopo do trabalho se resumisse à SPA, o banco de dados utilizado foi o Firebase [60], que oferece a possibilidade de comunicação em tempo real necessária para a aplicação.

4.3.1 Primeira iteração

Na primeira iteração, o sistema foi desenvolvido sem o uso de quaisquer padrões além do Componente Contâiner, explicado na Seção 3.3.1. O resultado está ilustrado na Figura 4.1.

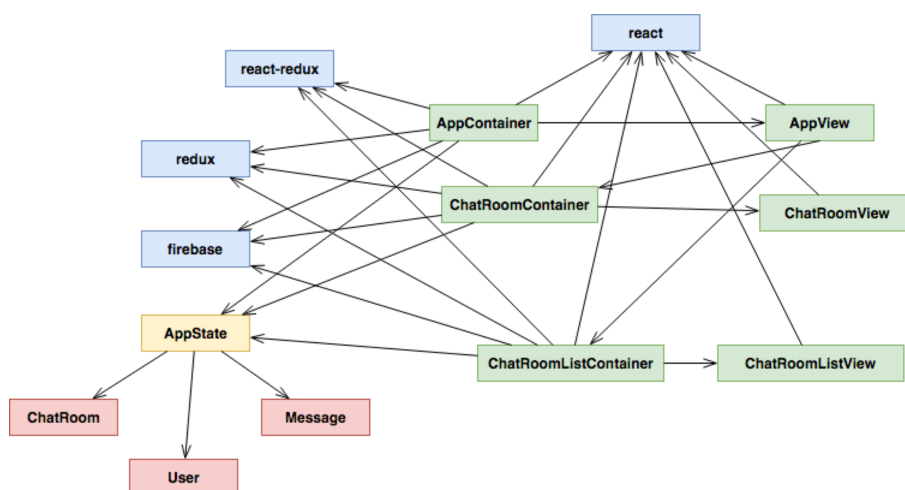

Figura 4.1 Visão geral das dependências entre os elementos que compõem a primeira iteração da aplicação. As cores usadas representam as camadas da Arquitetura limpa, ilustradas na Figura 3.6.

A arquitetura resultante é muito simples e pode ser suficiente no desenvolvimento de provas de conceito ou de projetos descartáveis. Mas nesse modelo, muito comum no estado da prática, há falhas óbvias: a dependência entre os Contâineres, o Firebase e o Redux são claras violações da Arquitetura Limpa e do Princípio da Inversão de Dependências. O mesmo vale para a dependência entre os Contâineres e o React. Além disso, os Contâineres violam o Princípio da Responsabilidade Única: são responsáveis pela execução de casos de uso, pela transformação de dados para as Views e pela ligação entre o estado da aplicação e a View.

É importante notar que dependência entre as Views e o React também denotam uma violação desses princípios, mas a correção desta é muito complexa para valer a pena: as bibliotecas de interface para Web se baseiam em princípios demasiadamente diferentes. Contornar essas diferenças causaria um problema citado em [45], a Complexidade Desnecessária.

4.3.2 Segunda iteração

Na segunda iteração, tentamos sanar os problemas enfrentados na primeira iteração de duas maneiras: aplicando a Inversão de Dependências nos elementos que acessam bibliotecas externas, e dividindo os Contâineres em três elementos com responsabilidades distintas.

A Inversão de Dependências foi aplicada através da criação de elementos que chamamos de Serviços. Os Serviços provém uma interface através da qual os controladores irão se comunicar de forma agnóstica aos detalhes de implementação. Dessa forma, poderíamos trocar o uso do Firebase, por exemplo, por uma outra solução de backend sem precisar modificar o código que utiliza os serviços.

Além disso, os Contâineres foram divididos em três elementos: Controladores, ViewModels e Módulos:

- **Controladores** tem a função de responder a interações do usuário, executando os casos de uso da aplicação;
- **ViewModels** tem a função de abstrair as entidades da aplicação, fornecendo à View apenas os dados formatados apropriadamente para exibição na interface;
- **Módulos** tem a função de instanciar Controladores, ViewModels e Views, agregando-os através de uma entidade a que chamamos de Montador de Módulo.

O Montador de Módulo é uma ideia que pode parecer um pouco menos comum ao cenário de desenvolvimento de SPAs. Ele é responsável por abstrair o código responsável por ligar as Views à solução de gerenciamento de estado, instanciando qualquer tipo de observação do estado necessária para a reatividade da aplicação. O resultado da iteração pode ser visto na Figura 4.2.

Nesse modelo, ainda temos elementos dependentes do Firebase e Redux, mas estes são elementos com o único propósito de agir como uma fronteira entre a aplicação e serviços externos. Isso faz com que mudanças nessas dependências afetem apenas um conjunto conhecido e muito bem definido de elementos, e como o resto do sistema depende apenas das interfaces realizadas por eles, não são necessárias alterações adicionais.

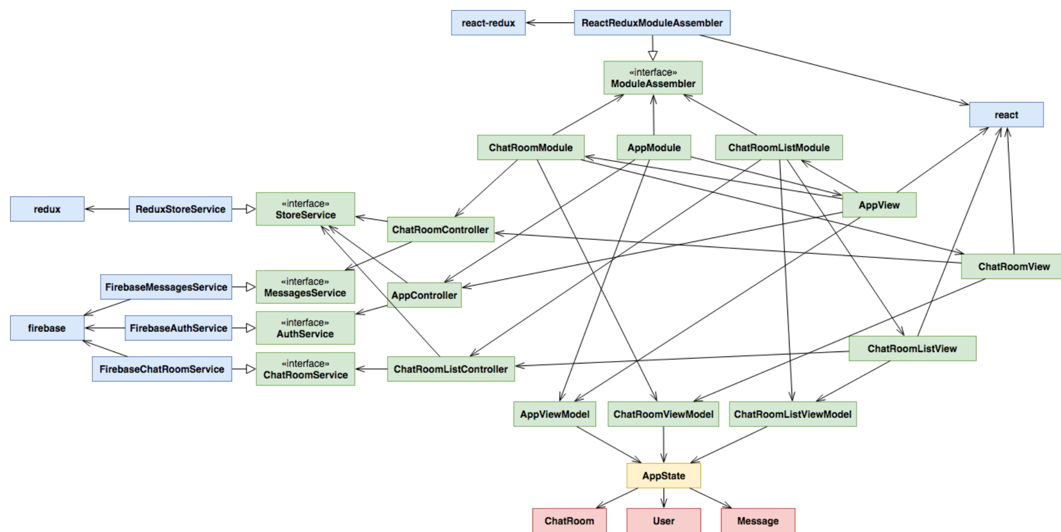

Figura 4.2 Visão geral das dependências entre os elementos que compõem a segunda iteração da aplicação.

4.4 Sumário do Capítulo

Nesse Capítulo, foi apresentada uma proposta de padrões arquiteturais a serem aplicados no projeto e desenvolvimento de SPAs, bem como o problema que esse conjunto de padrões se propõe a abordar. Foi também exibido um exemplo da aplicação de tais padrões, partindo de uma aplicação desenvolvida de forma simples, mas que reflete o estado da prática. No Capítulo seguinte, será demonstrada a maneira como esses padrões foram avaliados, bem como as conclusões tomadas acerca dessa análise.

CAPÍTULO 5

Análise

Sobre cada iteração descrita no Capítulo 4, foi feito um teste de modo a analisar o custo de mudanças de dependências externas sobre o projeto em termos da degradação de código [40] causada pelas alterações. De modo a medir esse custo, adotamos duas métricas: a quantidade de arquivos alterados e a razão entre a quantidade de linhas de código alteradas e a quantidade total de linhas de código do projeto.

O teste consistia em realizar a troca da ferramenta usada para gerenciamento de estado da aplicação. Inicialmente fazendo uso de Redux, a aplicação passaria a utilizar a biblioteca Mobx. Após realizada a mudança, as alterações de código foram medidas através das ferramentas de comparação oferecidas pelo git, sistema de controle de versões utilizado durante o desenvolvimento. Ao concluir a execução de um teste, o resultado foi medido através do comando

```
git diff --stat <start> <end>
```

em que *start* e *end* são os hashes que identificam o commit anterior ao início do teste e o commit que finalizou o teste, respectivamente. Foram analisados:

- A contagem de arquivos criados, deletados e modificados
- A contagem de linhas adicionadas e removidas do projeto em relação ao total de linhas do projeto. O total de linhas de código do projeto em cada iteração é apresentado na Tabela 5.1

Iteração	linhas de código
1	985
2	1130

Tabela 5.1 Total de linhas de código do projeto em cada iteração antes da realização dos testes. A contagem inclui apenas arquivos JavaScript no projeto, excluindo código CSS e HTML.

Com essas métricas, foi possível analisar de que maneira as alterações realizadas entre as iterações afetaram a base de código em relação a mudanças dependências externas. Os resultados aferidos estão representados nas Tabelas 5.3.

Iteração	arquivos criados	arquivos removidos	arquivos modificados
1	1	5	4
2	2	6	6

Tabela 5.2 Resultados do cenário de troca da biblioteca Redux pela biblioteca Mobx nas duas versões do sistema em termos de arquivos alterados.

Iteração	linhas adicionadas	linhas removidas	linhas modificadas
1	36	181	78
2	70	208	36

Tabela 5.3 Resultados do cenário de troca da biblioteca Redux pela biblioteca Mobx nas duas versões do sistema em termos de linhas de código.

Num primeiro momento, as mudanças nos números obtidos entre as iterações parecem irrisórias, apesar de ser notável uma diminuição na quantidade de código que foi modificado no projeto. Essa diminuição pode se tornar muito relevante num projeto maior, se levarmos em conta que alterações em código existente acarretam impacto negativo na mantenabilidade de um projeto [61]. Uma diferença muito maior pode ser percebida ao observar que tipo de arquivos foram modificados em cada iteração. As Figuras 5.1 e 5.2 demonstram em verde elementos que foram criados, e em vermelho elementos que foram removidos em cada iteração.

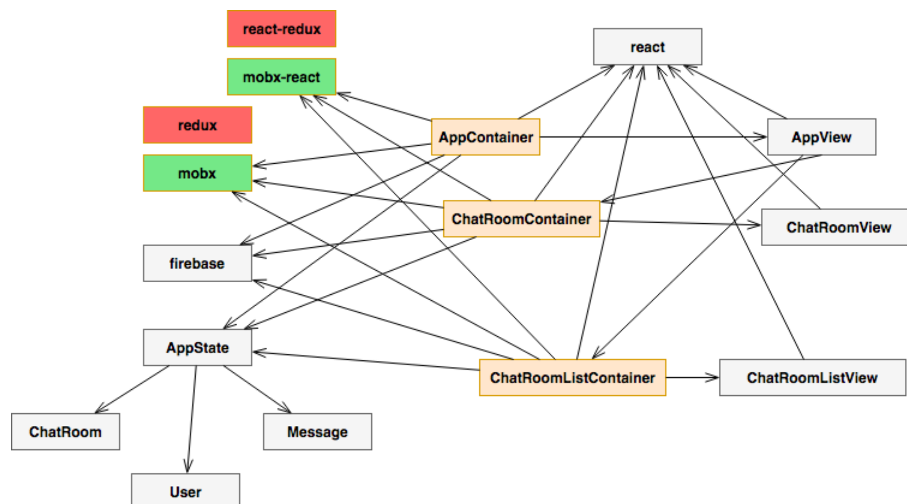

Figura 5.1 Elementos afetados pelo teste na primeira iteração.

Na primeira iteração, podemos ver que os elementos afetados pela execução do teste foram os Contâineres, enquanto na segunda iteração as mudanças se resumiram às fábricas dos serviços de operações sobre o estado da aplicação e de Montagem de Módulos. Isso traz uma implicação interessante: a quantidade de arquivos modificados na primeira iteração é de ordem linear em relação à contagem de Contâineres existentes na aplicação, enquanto na segunda iteração uma quantidade constante de arquivos são modificados. Esse resultado se mostra como um bom exemplo do Princípio Aberto/Fechado, explicado na Seção 3.2.2.

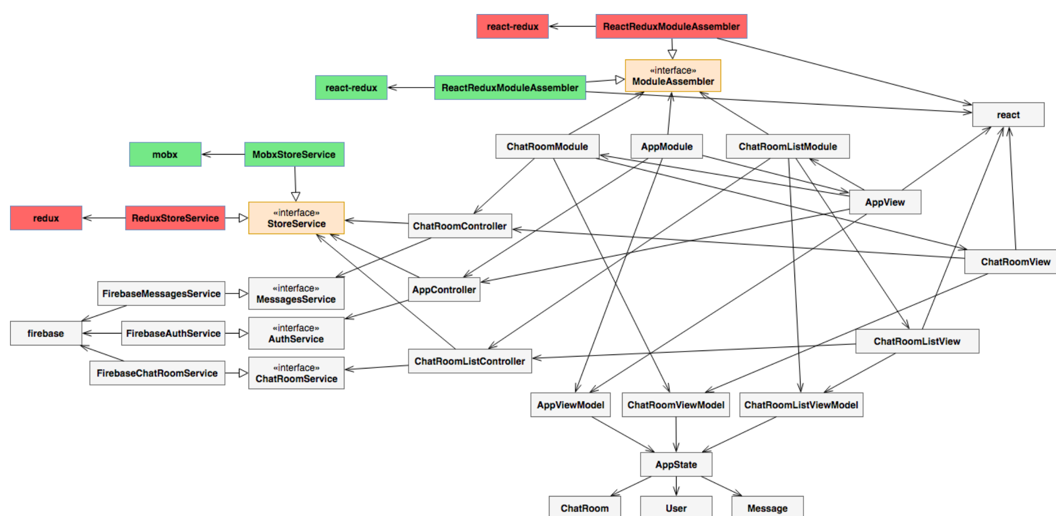

Figura 5.2 Elementos afetados pelo teste na segunda iteração. As interfaces em si não foram modificadas, mas estão marcadas como tal para simbolizar alterações feitas na função que instancia a implementação delas.

5.1 Sumário do Capítulo

Neste Capítulo, foi possível acompanhar uma tentativa de validação dos padrões propostos no Capítulo 4. Apesar de pouco significativos, os resultados podem ter revelado uma diminuição na modificação de código existente, diminuindo assim a degradação de código. Os resultados se tornam ainda mais interessantes quando os analisamos qualitativamente. No próximo Capítulo, concluiremos o trabalho e revelaremos possibilidades de trabalhos futuros que são capazes de trazer resultados mais incisivos e melhorias nos padrões apresentados.

CAPÍTULO 6

Conclusão e trabalhos futuros

A disponibilidade da Web como plataforma até então livre de uma entidade controladora da distribuição de seu conteúdo a torna atraente para muitos negócios. E com a evolução tecnológica dos navegadores Web de hoje, aplicações cada vez maiores e mais complexas do lado do cliente se tornam possíveis.

Em contrapartida, foi notado durante os estudos que alguns dos padrões utilizados no estado da prática não bastam para permitir que uma aplicação de grande porte seja mantida com facilidade e baixo custo. Procuramos por respostas em princípios criados há décadas, mas que ainda mostram sua eficácia durante o projeto de sistemas computacionais.

Este trabalho obteve resultados pouco conclusivos em relação às métricas utilizadas. No entanto, foi possível encontrar aplicação em princípios de arquitetura de software que permitem que aplicações mais complexas se tornem mais resilientes a mudanças. Isso é de suma importância para empresas que possuem aplicações Web como o Gmail ou o Facebook, com bases de código que chegam a milhões de linhas de código.

Existem múltiplos pontos de melhoria no resultado atingido pela segunda iteração desse trabalho. Estes pontos se tornam mais claros à medida que se tem casos de uso mais complexos e uma aplicação de maior porte. Um exemplo disso é a violação do Princípio da Responsabilidade Única, violado nos Controladores.

Na aplicação implementada, eles são responsáveis tanto por lidar com as interações do usuário como por executar os casos de uso. No entanto, se tivéssemos mais de um ponto da aplicação responsável pela execução do mesmo caso de uso, poderíamos começar a ter problemas para manter o funcionamento consistente do sistema.

O interesse por uma aplicação mais complexa ressalta um problema encontrado durante a realização deste trabalho. Realizar os testes se mostrou uma atividade complicada, dado que ela poderia requerer uma quantidade elevada de tempo caso o indivíduo não tivesse familiaridade com algumas das ferramentas envolvidas nos testes.

Referências Bibliográficas

[1] S. Overflow. (2017) Stack overflow developer survey 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017

[2] Github. Electron | build cross platform desktop apps with javascript, html, and css. [Online]. Available: https://electronjs.org/

[3] Facebook. React native | a framework for building native apps using react. [Online]. Available: https://facebook.github.io/react-native/

[4] N. Foundation. About | node.js. [Online]. Available: https://nodejs.org/en/about/

[5] Cylon.js - javascript framework for robotics, physical computing, and the internet of things using node.js. [Online]. Available: https://cylonjs.com/

[6] npm. [Online]. Available: https://www.npmjs.com/

[7] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE software*, vol. 18, no. 3, pp. 112–111, 2001.

[8] E. Magnusson and D. Grenmyr, "An investigation of data flow patterns impact on maintainability when implementing additional functionality," 2016.

[9] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[10] History of the web - world wide web foundation. [Online]. Available: https://webfoundation.org/about/vision/history-of-the-web/

[11] A short history of javascript. [Online]. Available: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[12] R. Sharp. (2010) What is a polyfill? [Online]. Available: https://remysharp.com/2010/10/08/what-is-a-polyfill

[13] jquery. [Online]. Available: https://jquery.com/

[14] J. J. Garrett. (2005, feb) Ajax: A new approach to web applications | adaptive path. [Online]. Available: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

[15] B. Eich. (2008, aug) Ecmascript harmony. [Online]. Available: https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html

[16] (2009) Ryan dahl: Node.js, evented i/o for v8 javascript - jsconf.eu - 2009. [Online]. Available: https://www.jsconf.eu/2009/speaker/speakers_selected.html

[17] R. Dahl. (2009) Ryan dahl: Original node.js presentation. [Online]. Available: https://www.youtube.com/watch?v=ztspvPYybIY

[18] Angular. [Online]. Available: https://angular.io/

[19] Ember.js - a framework for creating ambitious web applications. [Online]. Available: https://www.emberjs.com/

[20] React - a javascript library for building user interfaces. [Online]. Available: https://reactjs.org/

[21] Vue.js. [Online]. Available: https://vuejs.org/

[22] omcljs/om: Clojurescript interface to facebook's react. [Online]. Available: https://github.com/omcljs/om

[23] Reagent: Minimalistic react for clojurescript. [Online]. Available: https://reagent-project.github.io/

[24] Clojurescript. [Online]. Available: https://clojurescript.org/

[25] home. [Online]. Available: http://elm-lang.org/

[26] Babel · the compiler for writing next generation javascript. [Online]. Available: https://babeljs.io/

[27] npm-stat: babel-core. [Online]. Available: https://npm-stat.com/charts.html?package=babel-core\&from=2016-11-29\&to=2017-11-29

[28] Reconciliation - react. [Online]. Available: https://reactjs.org/docs/reconciliation.html

[29] javascript - can anyone explain the difference between reacts one-way data binding and angular's two-way data binding - stack overflow. [Online]. Available: https://stackoverflow.com/a/37566693

[30] Flux | application architecture for building user interfaces. [Online]. Available: https://facebook.github.io/flux/docs/in-depth-overview.html#content

[31] Read me · redux. [Online]. Available: https://redux.js.org

[32] Prior art · redux. [Online]. Available: https://redux.js.org/docs/introduction/PriorArt.html

[33] Middleware · redux. [Online]. Available: https://redux.js.org/docs/advanced/Middleware.html

[34] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[35] webpack module bundler. [Online]. Available: https://webpack.github.io/

[36] npm-stat: react. [Online]. Available: https://npm-stat.com/charts.html?package=react\&from=2017-01-01\&to=2017-11-22

[37] Welcome to the apache software foundation! [Online]. Available: https://www.apache.org/

[38] (2017) [legal-303] rocksdb integrations - asf jira. [Online]. Available: https://issues.apache.org/jira/browse/LEGAL-303?focusedCommentId=16088663\&page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel#comment-16088663

[39] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992.

[40] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[41] B. Len, C. Paul, and K. Rick, "Software architecture in practice," *Boston, Massachusetts Addison*, 2003.

[42] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[43] ——, “The dependency inversion principle,” *C++ Report*, vol. 8, no. 6, pp. 61–66, 1996.

[44] L. Hochstein and M. Lindvall, “Diagnosing architectural degeneration,” in *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*. IEEE, 2003, pp. 137–142.

[45] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[46] ——. (2012, aug) The clean architecture | 8th light. [Online]. Available: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

[47] T. DeMarco, *Structured analysis and system specification*. Yourdon Press, 1979.

[48] M. P.-J. Page-Jones, *The practical guide structured systems design*. Prentice-Hall,, 1988.

[49] B. Meyer, *Object-oriented software construction*. Prentice hall New York, 1988, vol. 2.

[50] Alistair.cockburn.us | hexagonal architecture. [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture

[51] S. Freeman and N. Pryce, *Growing Object-oriented Software: Guided by Tests*. Pearson Education India, 2009.

[52] State and lifecycle - react. [Online]. Available: https://reactjs.org/docs/state-and-lifecycle.html

[53] R. S. Pressman, “What a tangled web we weave [web engineering],” *IEEE Software*, vol. 17, no. 1, pp. 18–21, 2000.

[54] The cost of javascript - dev channel - medium. [Online]. Available: https://medium.com/dev-channel/the-cost-of-javascript-84009f51e99e

[55] P. Kruchten, “Mommy, where do software architectures come from,” in *Proceedings of the 1st Intl. Workshop on Architectures for Software Systems*, 1995, pp. 198–205.

[56] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns*. Elsevier, 2002.

[57] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Reading, 1996, vol. 335.

[58] Microsoft. Typescript - javascript that scales. [Online]. Available: https://www.typescriptlang.org/

[59] Facebook. Flow: A static type checker for javascript. [Online]. Available: https://flow.org/

[60] Google. Firebase. [Online]. Available: https://firebase.google.com/

[61] C. Faragó, "Variance of source code quality change caused by version control operations." *Acta Cybern.*, vol. 22, no. 1, pp. 35–56, 2015.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).