

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

BRUNO DE MORAIS BUENO

Análise de geração automatizada de código fonte: do protótipo ao software

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marcelo Soares Pimenta

Porto Alegre
2023

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos André Bulhões

Vice-Reitora: Prof.^a Patricia Pranke

Pró-Reitora de Graduação: Cíntia Inês Boll

Diretora do Instituto de Informática: Prof.^a Carla Maria Dal Sasso Freitas

Coordenador do Curso de Ciência da Computação: Prof. Marcelo Walter

Bibliotecário-Chefe do Instituto de Informática: Alexsander Borges Ribeiro

AGRADECIMENTOS

Agradeço aos meus pais Valdomiro e Eliane e meu irmão Pedro Luís pela compreensão, liberdade e apoio para meus estudos. Aos meus avós Antenor e Maria Atair por demonstrarem orgulho e me motivarem em minha jornada. A comunidade preta de tecnologia da informação que me orientou em minha jornada profissional e a toda comunidade do Instituto de Informática da UFRGS que me cedeu extremo conhecimento em computação, evolução profissional e pessoal. E a meu Prof. Marcelo Pimenta por sempre ser solícito e compreensivo na elaboração deste trabalho.

Meu sincero, muito obrigado.

RESUMO

O processo de desenvolvimento de software está cada vez mais aderente ao uso de tecnologias com enfoque na Experiência do Usuário (UX) e interação deste usuário com o sistema, estabelecendo maiores conexões das áreas de engenharia de software e design, principalmente nos processos de prototipagem desses sistemas com várias ferramentas disponíveis no mercado. Ao analisar as tendências no desenvolvimento de aplicações web há um grande foco na modularização de código, principalmente nas tecnologias de interface utilizadas, introduzindo o conceito de componentização, além disso, há também ferramentas para prototipação de software que abstraem deste mesmo conceito, permitindo a componentização do design desses sistemas. O foco deste trabalho é (1) analisar plugins da plataforma de prototipação de interface Figma que geram código fonte Javascript/ReactJS de forma autônoma e (2) realizar um comparativo entre 2 (duas) soluções presentes no mercado com base em critérios como facilidade de adoção e uso, e revisão do código gerado com base em alguns critérios de engenharia de software como organização e qualidade.

Palavras-chave: Prototipação, Engenharia de Software, componentização, JavaScript, Figma.

Automated source code generation analysis: from prototype to software

ABSTRACT

The software development process is increasingly adhering to the use of technologies with a focus on User Experience (UX) and user interaction with the system, establishing greater connections in the areas of software engineering and design, mainly in the prototyping processes of these systems with various tools available on the market. When analyzing trends in the development of web applications there is a big focus on code modularization, mainly in the interface technologies used, introducing the concept of componentization, in addition, there are also tools for software prototyping that abstract from this same concept, allowing design componentization of these systems. The focus of this work is (1) to analyze plugins from the Figma interface prototyping platform that generate Javascript/ReactJS source code autonomously and (2) to carry out a comparison between 2 (two) solutions present on the market based on criteria such as ease of adoption and use, and review of the generated code based on some software engineering criteria such as organization and quality.

Keywords: Prototyping, Software Engineering, componentization, JavaScript, Figma.

LISTA DE FIGURAS

Figura 1 – Exemplo regra CSS aplicada aos elementos com classes <i>button</i> e <i>large</i>	12
Figura 2 – Exemplo definição e uso de variável CSS	13
Figura 3 – Elemento de interface, estrutura de nó Figma representada em JSON	15
Figura 4 – Figma, variante de componente de interface	15
Figura 5 – Figma, instância de componente.....	17
Figura 6 – Tela Figma e janela plugin Clapy	19
Figura 7 – Tela Figma e janela plugin Anima.....	20
Figura 8 – Figma, caixa de seleção e suas variações.....	24
Figura 9 – Figma, campos de texto e suas variações.....	25
Figura 10 – Figma, botões e suas variações	26
Figura 11 – Figma, ícones, textos e suas variações.....	26
Figura 12 – Figma, interface formulário de cadastro	27
Figura 13 – Figma, interface formulário de login	27
Figura 14 – Tela inicial plugin Clapy, opções de ajuda e documentação	29
Figura 15 – Tela inicial plugin Clapy, itens passíveis de ações pelo usuário	30
Figura 16 – Tela de processamento Clapy	31
Figura 17 – Tela de conclusão de processamento Clapy.....	31
Figura 18 – Feedback de erro	32
Figura 19 – Arquivos do código gerado com as variações de campos de texto	33
Figura 20 – Clapy, código fonte gerado para campos de texto	34
Figura 21 – Código CSS gerado, campos de texto sem e com ícone	35
Figura 22 – Código Javascript gerado, campos de texto sem e com ícone	36
Figura 23 – Código Javascript gerado, duas variações de botões de ação	37
Figura 24 – Tela inicial plugin Anima, opções de ajuda e documentação	37
Figura 25 – Tela inicial plugin Anima, itens clicáveis.....	39
Figura 26 – Tela inicial plugin Anima, processamento geração de código fonte	40
Figura 27 – Tela inicial plugin Anima, processamento geração de código fonte	41
Figura 28 – Anima arquivos do código gerado com as variações de campos de texto	42
Figura 29 – Anima, código fonte Javascript e CSS gerado para campos de texto	43
Figura 30 – Anima, código fonte Javascript gerado para campos de texto.....	44
Figura 31 – Anima, código fonte CSS gerado para campos de texto.....	45
Figura 32 – Anima, código fonte Javascript gerado para botões de ação	46
Figura 33 – Anima, código fonte CSS gerado para os botões de ação.....	47

LISTA DE TABELAS

Tabela 7.1 – Síntese da análise de critérios de usabilidade.....	48
Tabela 7.2 – Linhas de código fonte (SLOC)	50
Tabela 7.3 – Número total de módulos e funções Javascript geradas	50
Tabela 7.4 – Número total de módulos ou classes CSS geradas	51

LISTA DE ABREVIATURAS E SIGLAS

UX	User Experience, ou experiência do usuário
HTML	HyperText Markup Language
XHTML	eXtensible Hypertext Markup Language
HTTP	HyperText Transfer Protocol
CSS	Cascading Style Sheets
XML	eXtensible Markup Language
W3C	World Wide Web Consortium
DOM	Document Object Model
API	Application Programming Interface
ECMA	European Computer Manufacturers Association
JSON	Javascript Object Notation
SLOC	Source Lines of Code
JSX	JavaScript Syntax Extension

SUMÁRIO

1 INTRODUÇÃO	9
2 FUNDAMENTAÇÃO TEÓRICA	11
2.1 Tecnologias e ferramentas	11
2.1.1 Web e aplicações Web	11
2.1.2 <i>HyperText Markup Language (HTML)</i>	11
2.1.3 <i>Cascading Style Sheets (CSS)</i>	11
2.1.4 <i>Document Object Model (DOM)</i>	13
2.1.5 Javascript (ECMAScript)	13
2.1.6 <i>ReactJS</i>	14
2.1.7 Figma	14
2.1.7.1 Estrutura de elementos de interface Figma	14
2.1.7.2 Elementos, suas variantes e instâncias	15
2.1.7.3 Plugins Figma	17
2.1.8 Sonarqube	18
2.1.9 Design Systems	18
2.1.10 Critérios ergonômicos de usabilidade	18
3 APLICAÇÕES ESTUDADAS	19
3.1 Clapy	19
3.2 Anima	19
4 CRITÉRIOS DE AVALIAÇÃO	21
4.1 Critérios de usabilidade	21
4.1.1 Opções de ajuda e documentação	21
4.1.2 Densidade informacional	21
4.1.3 Feedback Imediato	22
4.1.4 Ações mínimas	22
4.2 Critérios de resultado	22
4.2.1 Nomenclatura	22
4.2.2 Linhas de código fonte (<i>SLOC, Source Lines of Code</i>)	23
4.2.3 Número total de módulos, classes e funções	23
4.2.4 Critério de reuso	23
5 ELEMENTOS DE INTERFACE	24
5.1 Caixa de seleção (<i>Checkbox</i>)	24
5.2 Campos de texto (<i>Text Field</i>)	24
5.3 Botões de ação (<i>Action buttons</i>)	25
5.4 Ícones e textos	26
5.5 Elementos mistos	26
6 ANÁLISE DE PLUGINS FIGMA	28
6.1 Clapy	28
6.1.1 Da usabilidade	28
6.1.1.1 Opções de ajuda e documentação	28
6.1.1.2 Densidade informacional	29
6.1.1.3 Feedback Imediato	30
6.1.1.4 Ações mínimas	32
6.1.2 Do resultado final gerado	32
6.1.2.1 Nomenclatura	32
6.1.2.2 Linhas de código fonte (SLOC)	34
6.1.2.3 Número total de módulos, classes e funções	34

6.1.2.4 Critério de reuso	35
6.2 Anima.....	37
6.2.1 Da usabilidade.....	37
6.2.1.1 Opções de ajuda e documentação	37
6.2.1.2 Densidade informacional	39
6.2.1.3 Feedback Imediato	40
6.2.1.4 Ações mínimas.....	41
6.2.2 Do resultado final gerado.....	42
6.2.2.1 Nomenclatura.....	42
6.2.2.2 Linhas de código fonte (SLOC).....	43
6.2.2.3 Número de módulos, classes e funções.....	43
6.2.2.4 Critério de reuso	44
7 RESULTADOS OBTIDOS	48
7.1 Análise de usabilidade	48
7.2 Análise de resultado	49
7.2.1 Da nomenclatura	49
7.2.2 Da análise estática de código.....	49
7.2.3 Do reuso.....	51
8 CONCLUSÃO.....	52
9 REFERÊNCIAS	53

1 INTRODUÇÃO

Nas últimas décadas a área de redes de computadores vem evoluindo através de inovações tecnológicas em todas as frentes. Tudo se inicia com a evolução dos principais componentes que compõe uma aplicação *Web*, como o HTML (*HyperText Markup Language*), o protocolo HTTP (*HyperText Transfer Protocol*), servidores e os browsers (KUROSE, 2010) ao ponto de se tornar acessível ao público geral. Com estes avanços a *Web* proporcionou a criação de aplicações com várias finalidades, como envio de e-mails, comunicação, compartilhamento de dados, mas principalmente viabilizou o acesso facilitado a aplicações mais robustas.

A posterior evolução das tecnologias de aplicações *Web* provocou o surgimento da linguagem Javascript e do mecanismo CSS (*Cascading Style Sheets*), responsáveis pela manipulação, validação e estilização do HTML, o que facilitou o desenvolvimento dessas aplicações. Com o passar dos anos surgiram novos recursos para estes elementos, visando principalmente a melhoria da produção e manutenção dos softwares *web*, como *frameworks* e bibliotecas para o Javascript, técnicas para melhor processamento de CSS e melhorias e atualizações no HTML.

Vale ressaltar alguns recursos que atualmente impactam principalmente na área de desenvolvimento de software, os *frameworks* Javascript atuais utilizam-se de vários conceitos e conhecimentos destacados na engenharia de software, a exemplo, modularização e reaproveitamento de código. Além disso, houve o surgimento de várias ferramentas para prototipação de sistemas em geral, mas no caso de sistemas *web* algumas evoluíram ao ponto de abstrair a modularização também de elementos de interface. A possibilidade de unir e integrar esses recursos prevê uma grande melhoria no processo de desenvolvimento de um software, do design de interface até a codificação funcional de um sistema.

Esse trabalho tem como proposta analisar e comparar aplicações que geram código fonte automaticamente a partir de protótipos de interface de sistema *web*, em particular aplicações plugin complementares a ferramenta Figma que geram código fonte Javascript, mais precisamente utilizando *framework* ReactJS. A proposta principal é realizar uma avaliação usando como base critérios de usabilidade e de engenharia de software, a fim de elencar facilidade de uso e qualidade do código gerado.

Nesse contexto, será utilizada como ferramenta de prototipação o software Figma e 2 de seus plugins responsáveis pela conversão de elementos de uma interface para um código funcional em ReactJS, *framework* Javascript que tem como principal proposta a modularização de software através do conceito de componentização.

Para análise e argumentação dos itens estudados adota-se como processo metodológico uma pesquisa descritiva com uma abordagem qualitativa, com base na exploração das ferramentas e plugins propostos, além de, uma revisão bibliográfica de estudos e pesquisas na área de engenharia de software e usabilidade visando verificar e comparar as melhores soluções com base no processo de uso e o código gerado, sua organização e qualidade.

Da estruturação do trabalho, inicialmente no capítulo 2 serão apresentados os principais conceitos e referências utilizados, no capítulo 3 são apresentados sucintamente os plugins a serem avaliados e no capítulo 4 os critérios utilizados para a avaliação desses plugins. Após nos capítulos 5 e 6 são discutidas as observações coletadas na utilização e resultado do uso dos plugins Clapy e Anima respectivamente, conforme os critérios e por fim no capítulo 7 é apresentado o resultado das discussões e observações coletadas nas avaliações e no 8 são apresentadas as conclusões obtidas com base nesses estudos e discussões.

2 FUNDAMENTAÇÃO TEÓRICA

No capítulo são apresentados conceitos e ferramentas utilizadas durante a pesquisa com foco no desenvolvimento de interfaces de aplicações e tecnologias utilizadas principalmente em ambiente *Web* e que apresentam as características que enfatizam a relação entre elemento de interface de aplicação e a implementação de software que o representa.

2.1 Tecnologias e ferramentas

2.1.1 Web e aplicações Web

A *Web* é uma aplicação cliente-servidor que permite aos usuários obter documentos de servidores *Web* por demanda. A aplicação *Web* consiste em muitos componentes, entre eles um padrão para formato de documentos, sendo definido como HTML, navegadores e servidores *Web*, e um protocolo de camada de aplicação (KUROSE, 2010).

Sendo assim, uma página *Web* é um arquivo codificado em *eXtensible Hypertext Markup Language* (XHTML ou HTML) ou uma variante do mesmo, *Cascading Style Sheets* (CSS) para definições de estilo/design dos elementos HTML, e pode ainda incorporar instruções da linguagem de programação Javascript, o que permite a manipulação e o processamento desses elementos (SAWAYA, 1999).

2.1.2 *HyperText Markup Language* (HTML)

O HTML é uma linguagem que permite criar aplicações que trabalham com textos e imagens numa mesma tela simultaneamente (SAWAYA, 1999).

O termo “Hipertexto” (tradução nossa) refere-se aos links que conectam as páginas *Web* entre si, enquanto, “Marcação” (tradução nossa) refere-se ao fato de que anota, delimita texto, imagem e outros conteúdos para exibição em um navegador *Web*. A “marcação” proposta pela linguagem inclui “elementos” especiais que em um mesmo documento são separados por *tags* (nomes dos elementos separados por “<” e “>”) (MOZILLA, 2023e, tradução nossa).

2.1.3 *Cascading Style Sheets* (CSS)

É uma linguagem de estilo utilizada para descrever e apresentar como os elementos contidos em um documento escrito em HTML ou em XML (*eXtensible Markup Language*) são renderizados na tela ou em outra mídia na qual é acessado, este é padronizado em navegadores

web de acordo com as especificações da W3C (*World Wide Web Consortium*). Basicamente, o CSS é baseado em regras que são especificadas para grupos de estilos e são aplicadas em elementos HTML particulares ou grupo de elementos de uma página web (MOZILLA, 2023a, tradução nossa).

Uma regra CSS define um aspecto de estilo de um ou mais elementos HTML e é composta por um seletor que especifica qual ou quais elementos possuirão determinada característica e por uma ou mais declarações que identificam de fato quais aspectos visuais são aplicados, já uma folha de estilo (*stylesheet*) é um conjunto de uma ou mais regras que se aplicam a um documento HTML (LIE; BOS, 1999).

Um seletor CSS é um trecho de código capaz de descrever atributos de elementos HTML para que sejam selecionados e que tenham propriedades CSS aplicadas, dentre esses seletores vale destacar o seletor de classe e de *id*, identificados com um “.” (ponto) e com “#” (*hashtag*) respectivamente, seguidos de um valor para identificar os elementos a serem selecionados no documento HTML (MOZILLA, 2023b, tradução nossa), conforme exemplo na Figura 1.

Figura 1 – Exemplo regra CSS aplicada aos elementos com classes *button* e *large*

```
.button.large {  
  min-height: 47px;  
  padding: 10px 15px;  
}
```

O CSS ainda conta com propriedades personalizadas (variáveis CSS) que são regras definidas que contêm valores específicos a serem reutilizados em um documento, estes são definidos usando a notação de propriedade personalizada (MOZILLA, 2023g) que pode ser visualizada na Figura 2 onde há a definição de uma variável (linha 2) e logo após seu uso em outra regra CSS (linha 6) definindo a cor de fonte para um elemento de parágrafo.

Figura 2 – Exemplo definição e uso de variável CSS

```
1  :root {  
2    --main-color: black;  
3  }  
4  
5  p {  
6    color: var(--main-color);  
7  }
```

2.1.4 Document Object Model (DOM)

É a representação de uma página web de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo. O DOM representa um documento com nós e objetos, definindo os chamados elementos que compõe a tela ou interface de uma aplicação web. Pode ainda ser definido como a representação orientada a objetos de uma página web e que pode ser modificada com uma linguagem de script como Javascript (MOZILLA, 2023c).

2.1.5 Javascript (ECMAScript)

É uma linguagem de programação utilizada principalmente para scripts dinâmicos do lado do cliente em páginas web, podendo também ser utilizada no lado do servidor, usando um interpretador como o Node.js. Esta permite a manipulação de conteúdo de uma página web por meio do DOM, a manipulação de dados comuns a uma linguagem de programação, comunicação com outros sistemas, operações em bancos de dados e etc., e com isso, permite o desenvolvimento de softwares robustos e de fácil acesso. (MOZILLA, 2023f)

Um navegador da Web fornece um ambiente de host ECMAScript para computação do lado do cliente, e juntamente com o DOM inclui objetos que representam janelas, menus, pop-ups, caixas de diálogo, áreas de texto, âncoras, quadros, histórico, cookies e entrada/saída. Além disso, o ambiente host permite e reconhece o disparo de eventos, como mudança de foco, carregamento de página e imagem, descarregamento, erro e interrupção, seleção, envio de formulário e ações do mouse que interagem com o código a ser executado, tornando a aplicação principalmente reativa à interação do usuário. (ECMA INTERNATIONAL, 2023).

O Javascript possui tipagem dinâmica com tipos comuns a outras linguagens de programação como *string*, *number*, *boolean* e *object* (ECMA INTERNATIONAL, 2023), e permite inclusive operações com valores de tipagens diferentes como, por exemplo, a concatenação de um valor *string* com um valor booleano transformando o valor booleano na respectiva *string true* ou *false*.

2.1.6 ReactJS

ReactJS é uma biblioteca JavaScript para construção de interfaces de usuário (META PLATFORMS, 2023), ele permite a criação de interfaces a partir de peças individuais chamadas de componentes que são nada menos que funções Javascript que retornam como resultado elementos que compõe o DOM da aplicação, possuem estado e podem ainda responder a interações de um usuário atualizando a tela conforme necessário (META OPEN SOURCE, 2023).

Códigos em ReactJS são desenvolvidos na extensão JSX (*JavaScript Syntax Extension*) com sintaxe do Javascript e que após compilados retornam objetos simples chamados de “elementos React” que correspondem a elementos do DOM (META OPEN SOURCE, 2023).

2.1.7 Figma

É uma ferramenta para prototipação de software portátil entre sistemas (*web*, Windows, MacOS e Linux) que possui salvamento automático em nuvem e permite a criação de *design systems* com componentes e estilos de interface vinculados entre si, reutilizáveis em projetos e protótipos facilmente (FIGMA, 2023a, tradução nossa).

Além disso, o Figma possui uma API (*Application Programming Interface*) com suporte a leitura e interação com arquivos desenvolvidos dentro da ferramenta, permitindo a visualização e extração de qualquer objeto com suas propriedades, de forma a representar os elementos de interface como uma estrutura de dados (FIGMA, 2023b, tradução nossa). O Figma ainda abstrai o conceito de componente, o usuário pode definir elementos de interface como componentes e reutilizá-los em seus designs e por toda uma equipe ao serem publicados. Botões, ícones, campos para inserção de dados, etc. são reaproveitados por todo o protótipo e acessíveis através da API da ferramenta (FIGMA, 2023b, tradução nossa).

2.1.7.1 Estrutura de elementos de interface Figma

No Figma os elementos de interface têm seus dados representados na forma de árvores de nós, a hierarquia principal inicia no nodo raiz *Document* referente ao arquivo Figma, seus filhos são as páginas do arquivo e em cada página há nós de tela e suas subárvores que representam um frame ou objeto em tela (FIGMA, 2023b, tradução nossa).

Os nós possuem diversas propriedades associadas sendo globais ou específicas a cada um, estas descrevem características de estilo para cada elemento em tela e que podem ser obtidas através de API no formato JSON (*Javascript Object Notation*), conforme Figura 3.

Figura 3 – Elemento de interface, estrutura de nó Figma representada em JSON

```

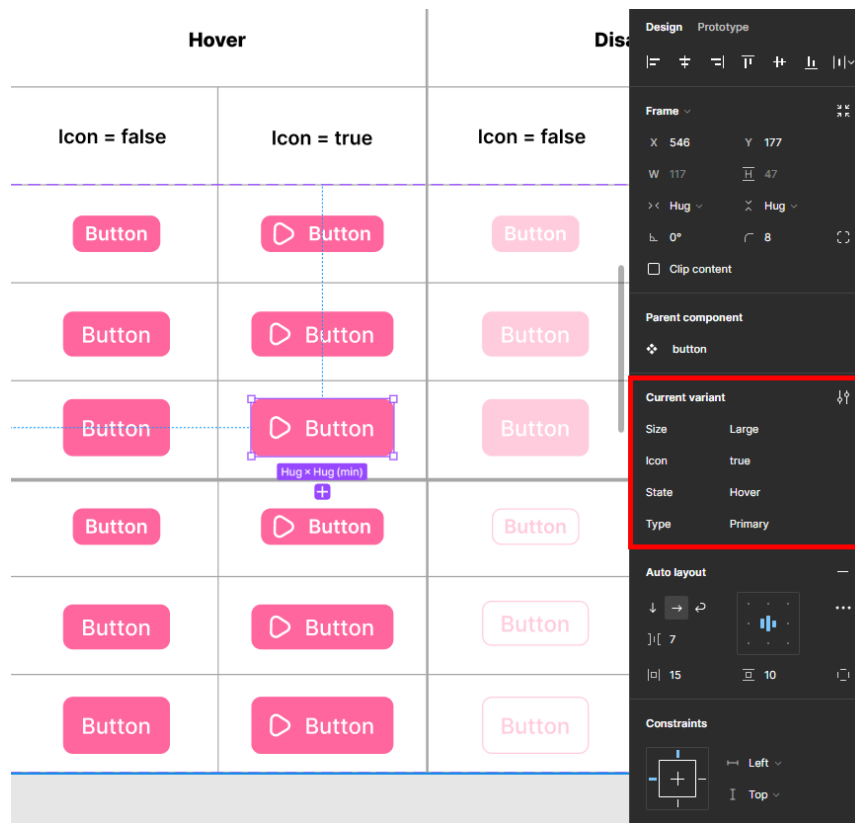
9      "nodes": {
10         "405:133": {
11             "document": {
12                 "id": "405:133",
13                 "name": "button/large/disable",
14                 "type": "COMPONENT",
15                 "scrollBehavior": "SCROLLS",
16                 "blendMode": "PASS_THROUGH",
17                 "children": [],
18             > "absoluteBoundingBox": { ...
23             },
24             > "absoluteRenderBounds": { ...
29             },
30             > "constraints": { ...
33             },
34             "layoutSizingHorizontal": "HUG",
35             "layoutSizingVertical": "HUG",
36             "clipsContent": false,
37             > "background": [ ...
48             ],
49             > "fills": [ ...
60             ],
61             "strokes": [],
62             "cornerRadius": 8.0,
63             "cornerSmoothing": 0.0,
64             "strokeWeight": 1.0,
65             "strokeAlign": "INSIDE",
66             "backgroundColor": {
67                 "r": 1.0,
68                 "g": 0.80000001192092896,
69                 "b": 0.8718954324722290,
70                 "a": 1.0
71             },
72             "styles": {
73                 "fills": "203:32",
74                 "fill": "203:32"
75             },
76             "layoutMode": "HORIZONTAL",

```

2.1.7.2 Elementos, suas variantes e instâncias

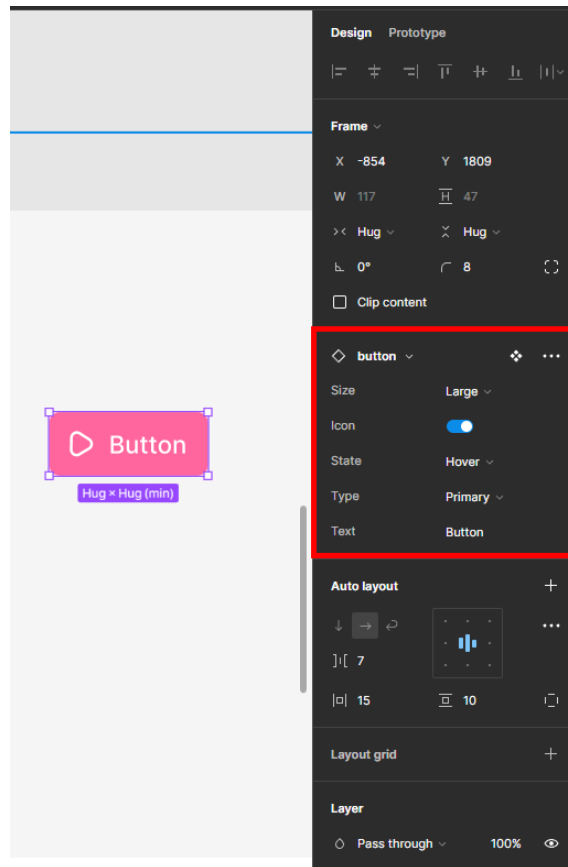
O Figma permite a definição de seus elementos de interface como componentes, ou conjunto de componentes, além da possibilidade de agrupá-los conforme suas semelhanças, chamados de variantes (FIGMA, 2023d, tradução nossa). Um componente no Figma possui propriedades e valores e a cópia, modificação do componente e definição de valores diferentes geram o que é chamado de componente variante, conforme Figura 4 há uma variante do componente botão com conjunto de atributos e valores exclusivos somente a ele e identificados pela indicação em vermelho.

Figura 4 – Figma, variante de componente de interface



Criados componentes e suas variantes pode-se copiá-los novamente gerando uma instância daquela variante, os valores de suas propriedades podem ser alterados o que altera suas características de forma a corresponder com alguma variante já definida. Conforme Figura 5 há uma instância com propriedades que a identificam como a variante demonstrada na Figura 4, a alteração desses valores faz com que o botão assumo o estilo da variante que corresponde aos novos valores.

Figura 5 – Figma, instância de componente



2.1.7.3 Plugins Figma

São scripts ou aplicações de terceiros que estendem as funcionalidades de produtos Figma interagindo através da API disponível, com acesso aos arquivos do usuário para leitura e gravação.

Além disso, plugins Figma podem usar APIs externas, visualizar, modificar e criar objetos ou elementos dentro de um arquivo na aplicação Figma (FIGMA, 2023c, tradução nossa), a exemplo, um elemento de interface criado em um arquivo pode ser consultado e seus dados ou características convertidos para outra representação diferente da original, como código funcional em Javascript.

Essas aplicações serão os objetos de estudo desse trabalho, especificamente os plugins que geram código funcional em ReactJS a partir de elementos de interface prototipados no Figma.

2.1.8 Sonarqube

É uma ferramenta de revisão automática de código capaz de realizar uma análise estática com base em critérios como limpeza de código seguindo as boas práticas, segurança e até coleta de dados referentes ao tamanho, repetições, bugs e modificações com uma estratégia de versionamento. Neste trabalho é utilizado para análise do código Javascript gerado pelos plugins Figma e coleta dos dados como número de linhas e número de funções declaradas.

2.1.9 Design Systems

É uma biblioteca de componentes e diretrizes reutilizáveis que as pessoas de uma empresa podem combinar em interfaces e interações, além de, fornecer diretrizes consistentes de estilo e interação para as equipes (INTERACTION DESIGN FOUNDATION, 2023, tradução nossa).

2.1.10 Critérios ergonômicos de usabilidade

É um conjunto de oito critérios ergonômicos principais que se subdividem em 18 subcritérios e critérios elementares definidos por Cybis, Betiol e Faust (2010) com o objetivo de minimizar a ambiguidade na identificação e classificação das qualidades e problemas ergonômicos de um software interativo.

Para as avaliações foram utilizados apenas 4 critérios referentes principalmente a condução do usuário e carga de trabalho do usuário sendo eles: Convite com foco em Opções de ajuda e documentação, densidade informacional, *feedback* imediato e ações mínimas.

Os principais pontos que levaram a definição de somente esses 4 critérios são o fato de que as aplicações estudadas são simples e tem somente 1 fluxo curto de utilização, não fazendo sentido avaliar os demais, além disso, o foco do estudo é a utilização repetida do plugin considerando a conversão de um *design system* robusto, com vários agrupamentos de variações de elementos.

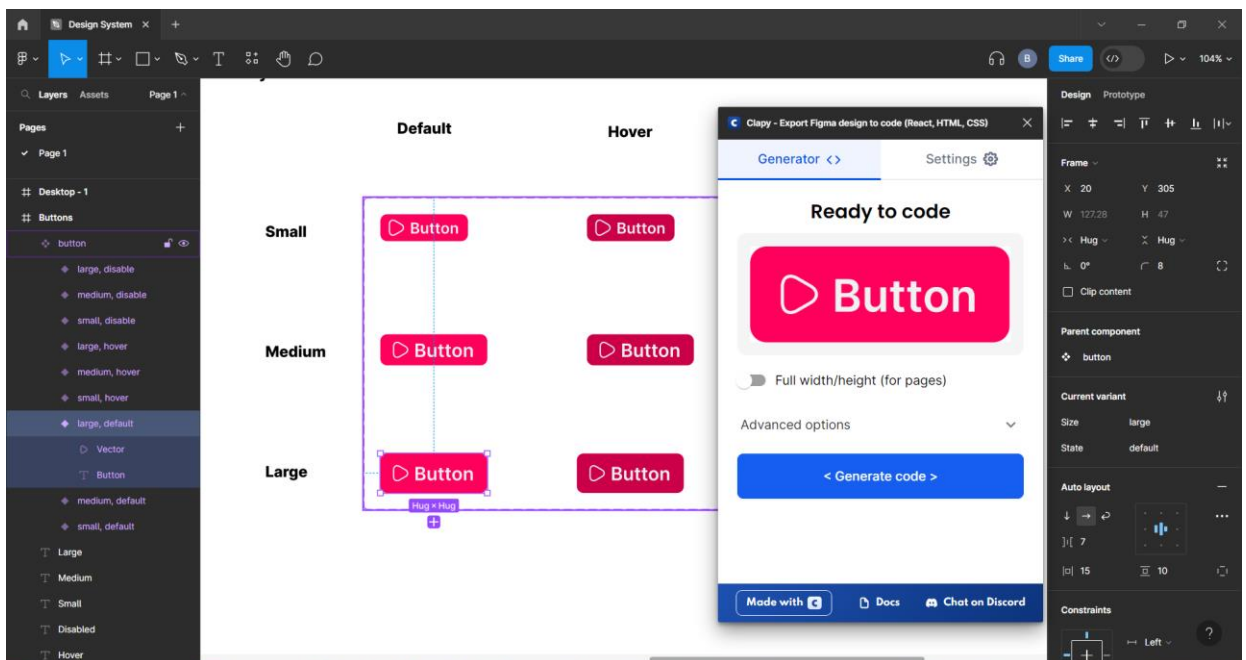
3 APLICAÇÕES ESTUDADAS

Neste capítulo são apresentadas soluções que permitem a conversão de um protótipo desenvolvido através da ferramenta Figma e o software o qual esta representa. Atualmente o mercado carece de ferramenta com uma integração contínua entre essas tecnologias, mas possui soluções que facilitam a codificação de um elemento de interface para um bloco de código estático, no geral tratam-se de plugins acoplados dentro da aplicação Figma e que serão mencionados a seguir.

3.1 Clapy

Plugin Figma que permite a geração de componentes ReactJS a partir de designs Figma, além disso, tem como proposta permitir a automatização de integração entre um protótipo e o código que o implementa e usar as melhores práticas de designers e desenvolvedores junto a ferramenta Figma (CLAPY, 2023).

Figura 6 – Tela Figma e janela plugin Clapy

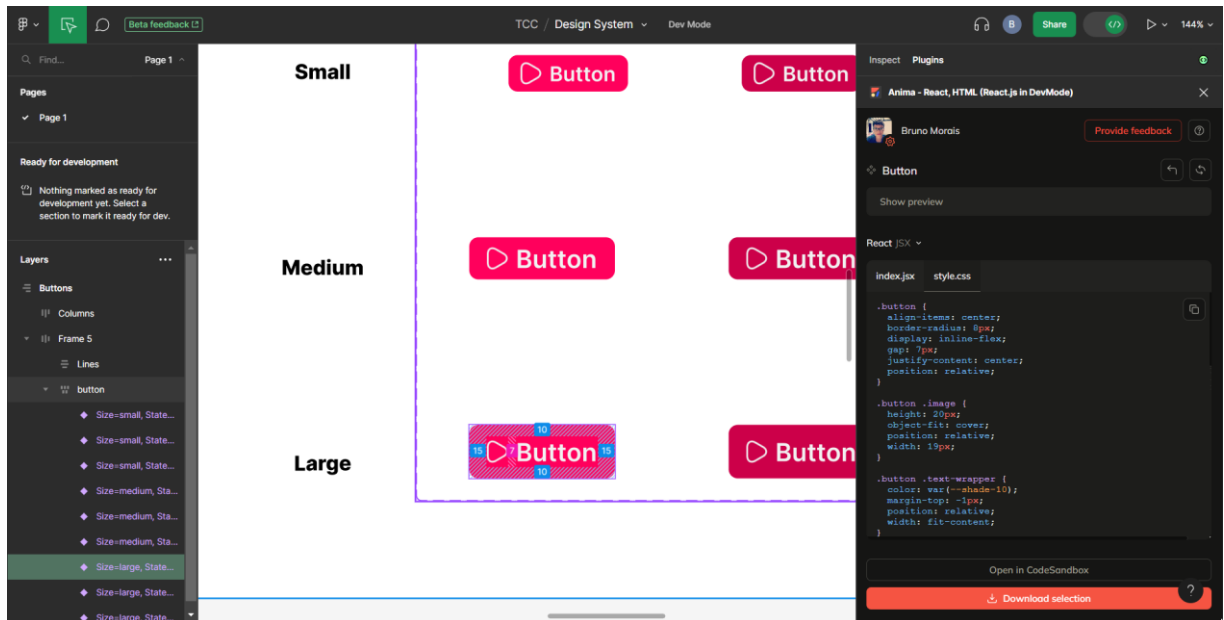


Fonte: Figma

3.2 Anima

É uma aplicação plugin com o objetivo de automatizar o processo de design para código funcional de uma aplicação, gerar código ReactJS/HTML limpo, funcional e de alta qualidade. A solução ainda permite a revisão, inspeção e cópia do conteúdo gerado diretamente dentro da aplicação Figma, visualização em tempo real e exportação do código para execução local ou no ambiente que o usuário preferir. (ANIMA, 2023, tradução nossa)

Figura 7 – Tela Figma e janela plugin Anima



Fonte: Figma

4 CRITÉRIOS DE AVALIAÇÃO

Critérios utilizados para análise comparativa de plugins do software Figma, levando em consideração a usabilidade e aspectos do resultado gerado. Visam justificar qual aplicação é mais efetiva e de fácil uso e aplicação em um processo de desenvolvimento de software.

4.1 Critérios de usabilidade

São os critérios referentes a utilização de cada plugin baseados nos princípios ergonômicos para Interação Humano-Computador (IHC) (CYBIS; BETIOL; FAUST, 2010), levando em consideração a simplicidade de um software complementar e que possui uma única tarefa. Os princípios ergonômicos visitados no trabalho têm foco no processo de uso, condução do usuário, ajuda e documentação.

4.1.1 Opções de ajuda e documentação

Baseado na qualidade elementar Convite do princípio de Condução, diz respeito ao acesso facilitado a instruções de uso e busca de ajuda ou documentação da aplicação (CYBIS; BETIOL; FAUST, 2010), nesse trabalho as aplicações serão classificadas em:

- Suficientes: Indica fácil acesso a suas instruções de uso e ajuda e fácil acesso a documentação, com informações que descrevem o processo de uso e que prevê erros nesse processo.
- Parciais: Indica fácil acesso a instruções de uso e ajuda e fácil acesso a documentação, porém, informações não descrevem instruções de uso efetivas ou há erros ou situações inesperadas não documentadas que atrasam ou bloqueiam a utilização.
- Insuficientes: Indica que não há fácil acesso a instruções de uso e ajuda e fácil acesso a documentação ou não há documentação.

4.1.2 Densidade informacional

Baseado no critério que diz respeito à carga de trabalho do usuário, de um ponto de vista perceptivo e cognitivo, é relacionado ao conjunto total de itens de informação apresentados (CYBIS; BETIOL; FAUST, 2010). Nesse trabalho será utilizado para indicar numericamente itens ou elementos da interface que são irrelevantes para a atividade principal do plugin, elementos desnecessários que oneram a densidade de informações para o usuário.

4.1.3 Feedback Imediato

Critério baseado na qualidade de mesmo nome presente no princípio de Condução, visa mensurar a fluidez do processo de utilização, se há *feedback* de processamento para processos mais longos (CYBIS; BETIOL; FAUST, 2010) e de erros em caso de falhas, os plugins podem ser classificados como:

- Satisfatório: Indica processo de execução de atividade fluido, processos mais longos possuem feedback de processamento, erros são imediatamente informados e bem descritos.
- Insatisfatório: Indica um processo não fluído e falta de feedback ao usuário, aplicação não funciona, há bloqueios no processo e não há clareza em como resolvê-los.

4.1.4 Ações mínimas

Qualidade que caracteriza o software que minimiza e simplifica um conjunto de ações necessárias para o usuário alcançar uma meta ou realizar uma tarefa (CYBIS; BETIOL; FAUST, 2010). Nesse trabalho é utilizada como métrica numérica indicando o número de ações que o usuário deverá executar para chegar ao resultado final.

4.2 Critérios de resultado

São os critérios referentes ao resultado da execução de cada plugin para gerar código funcional em ReactJS com base em elementos de interface de um arquivo Figma. Os plugins terão como entrada diferentes elementos de interface em um protótipo Figma, gerando diferentes implementações de código que podem ser analisadas com base na nomenclatura e estrutura do código fonte, como tamanho e reuso.

4.2.1 Nomenclatura

Tem o propósito de avaliar a concordância do padrão de nomes gerados pelo plugin, englobando a nomeação de arquivos, variáveis, constantes e propriedades. Neste trabalho a avaliação desse critério se dá utilizando um guia para escrita de código Javascript definida por Mozilla (2023d, tradução nossa) e considerando um padrão igual para todos os nomes gerados sendo ele *camelCase* ou *PascalCase*.

4.2.2 Linhas de código fonte (SLOC, *Source Lines of Code*)

É referente ao resultado da contagem de apenas linhas executáveis, são excluídas linhas em branco e comentários, para comparações entre sistemas usando SLOC é necessário que ambos tenham sido feitos na mesma linguagem de programação e que o estilo esteja normalizado (MEIRELLES, 2013). Nesse trabalho não há normalização do código gerado automaticamente, porém, a proposta é comparar a eficiência do processo de geração direta e autônoma, a normalização faz parte desse processo, além disso, para a efetiva contagem das linhas de código fonte geradas será utilizado o software Sonarqube.

4.2.3 Número total de módulos, classes e funções

É outro indicador de tamanho com o objetivo de identificar as declarações dos componentes de interface convertidos para código em forma de funções ou objetos, a fim de detectar melhores simplificações e reusos de código. Para este trabalho a contagem de funções Javascript se dará através da análise realizada pelo software Sonarqube e contagem de módulos ou declarações CSS será realizada pelo próprio autor.

4.2.4 Critério de reuso

Tem o objetivo de avaliar o código gerado com base no reuso de suas classes e funções, estruturando os componentes de software da forma mais ótima possível para as entradas dadas. Os elementos de interface que contém características ou atributos semelhantes em suas variações são passíveis de reutilização, visto que as definições estruturais de cada elemento podem ser compartilhadas, no caso definições de estilo em código CSS e de hierarquia dos elementos em código Javascript.

Neste trabalho serão usados alguns conjuntos de elementos de interface que possuem variações com atributos iguais, como conjunto de botões de ação e de campos de texto que tem, por exemplo, altura, largura e cores iguais, visando qualificar os plugins com base no reuso das definições desses atributos.

5 ELEMENTOS DE INTERFACE

Capítulo que descreve os elementos de interface simples e mais complexos utilizados como parâmetro de entrada para as aplicações plugin estudadas. São definidos componentes da interface a serem convertidos para código fonte na forma de grupos de elementos de mesmo tipo conforme dispostos em um Design System e na forma de composição de telas de um protótipo de exemplo, como tela simples de cadastro e de login.

Para esse trabalho foram usados conjuntos de elementos comuns a interfaces de aplicações web conforme definições de Garret (2010, tradução nossa):

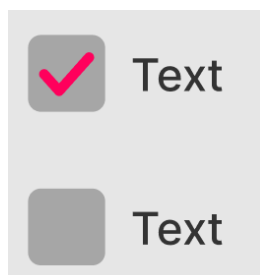
- Caixa de seleção (*Checkbox*): permite que usuários selecionem opções independentemente de outras.
- Campos de texto (*Text Fields*): permitem que usuários digitem textos como entrada através do teclado.
- Botões de ação (*Action Buttons*): como um gatilho ou alavanca indicam ao sistema para fazer algo, realizar alguma ação na aplicação.

Além de textos, ícones e caixas para separação de informações, como caixas de diálogo e algumas variações.

5.1 Caixa de seleção (*Checkbox*)

Componente simples com 2 variações: assinalada e não assinalada, variável *checked* com valor *true* ou *false*, respectivamente.

Figura 8 – Figma, caixa de seleção e suas variações



5.2 Campos de texto (*Text Field*)

Elemento com 4 variações, com rótulo e sem rótulo e com ícone e sem ícone, sendo as variáveis:

- Label: com valores *true* ou *false*, indicando presença ou não de *label* identificadora do campo.
- Icon: com valores *true* ou *false*, indicando presença ou não de ícone ao lado do espaço para digitação pelo usuário.

Figura 9 – Figma, campos de texto e suas variações



5.3 Botões de ação (*Action buttons*)

Elemento com 36 variações e 4 variáveis sendo elas:

- *Size*: referente a características de tamanho do botão, com valores *Small*, *Medium* e *Large*, indicando relação de tamanho pequeno, médio e grande respectivamente.
- *Icon*: com valores *true* ou *false*, indicando presença ou não de ícone interno ao botão.
- *State*: referente ao estado do botão, com valores *Default*, *Hover* e *Disabled*, indicando estado padrão, com presença de mouse sobre o botão e desabilitado, respectivamente.
- *Type*: referente ao tipo de ação principal ou secundária dado o contexto de uso do botão, com valores *Primary* e *Secondary*.

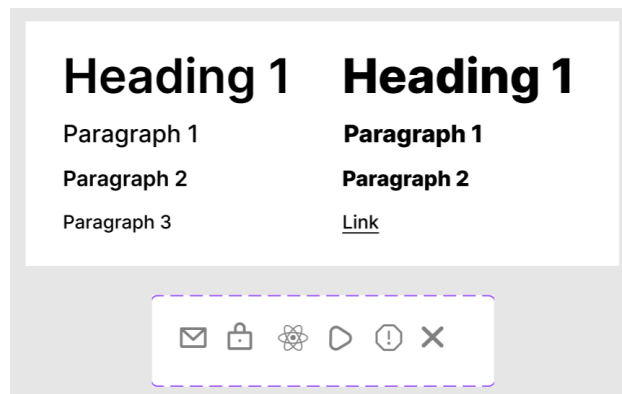
Figura 10 – Figma, botões e suas variações

	Default		Hover		Disabled	
	Icon = false	Icon = true	Icon = false	Icon = true	Icon = false	Icon = true
Small						
Medium						
Large						
Small						
Medium						
Large						

5.4 Ícones e textos

Elementos simples, sendo o texto uma caixa de texto preenchida com algum caracter e ícones sendo variações de conjuntos de formas geométricas a fim de caracterizar um ícone de interface, possuindo 6 variações.

Figura 11 – Figma, ícones, textos e suas variações



5.5 Elementos mistos

Elementos construídos com mais de um elemento variado em conjunto, como um formulário para preenchimento de campos com ação “Salvar” e um formulário para preenchimento e realização de um login, unindo vários elementos acima citados:

Figura 12 – Figma, interface formulário de cadastro

→ Cadastro

Nome completo:	E-mail:	
<input type="text" value="Seu nome"/>	<input type="text" value="johndoe@example.com"/>	
Login:	Data de nascimento:	Telefone:
<input type="text" value="seu.login"/>	<input type="text" value="xx/xx/xxxx"/>	<input type="text" value="(xx) xxxxx-xxxx"/>

Senha:	Confirmar senha:	
<input type="password" value="*****"/>	<input type="password"/>	
		<input type="button" value="Cancelar"/>
		<input type="button" value="Salvar"/>

Figura 13 – Figma, interface formulário de login



ReactJS
Faça login e comece a usar!

Endereço de e-mail:

Senha:

Lembrar de mim por 30 dias

[Não possui conta? Crie uma agora](#)

6 ANÁLISE DE PLUGINS FIGMA

Capítulo responsável por descrever e avaliar o processo de conversão de conjuntos de elementos de interface Figma para código estático por cada plugin proposto, levando em consideração os critérios discutidos no capítulo 4.

Para análise de resultado gerado foram utilizados como entrada 4 conjuntos diferentes de elementos de interface, sendo eles:

- Campos de texto e suas variações (Figura 9)
- Botões de ação e suas variações (Figura 10)
- Formulário de login (Figura 13)
- Formulário de cadastro (Figura 12)

6.1 Clapy

6.1.1 Da usabilidade:

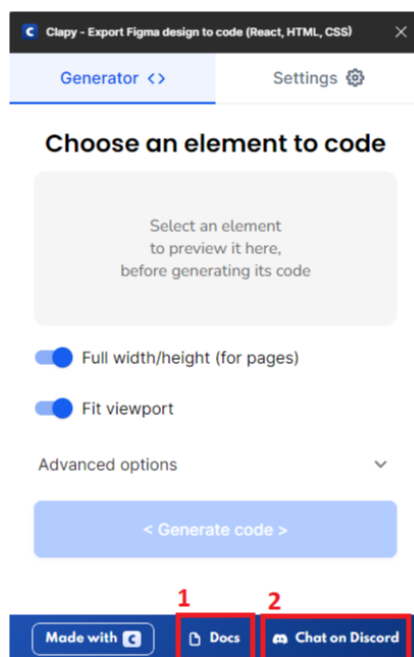
6.1.1.1 Opções de ajuda e documentação

Conforme janela da interface inicial do Clapy (Figura 14) há 2 meios de acesso a opções de ajuda e instruções sendo elas:

- *Docs*: Link de acesso a documentação completa do Plugin que inicia com uma introdução a aplicação, possui explicações aprofundadas em funcionalidades do Figma e até instruções das melhores práticas de criação de itens de interface, para que seja gerado um código fonte mais efetivo.
- *Chat on Discord*: Link de acesso a um servidor no Discord com canais para introdução e ajuda, reporte de bugs, avisos de atualizações e sugestões de funcionalidades.

Assim, o plugin é classificado como possuindo opções de ajuda e documentação suficientes conforme definição do critério discutido nessa seção.

Figura 14 – Tela inicial plugin Clapy, opções de ajuda e documentação



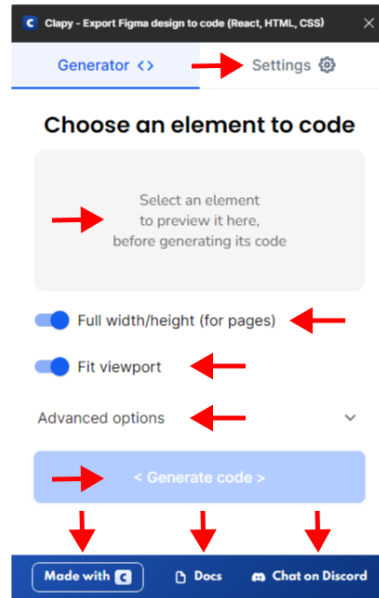
Fonte: Clapy (2023)

6.1.1.2 Densidade informacional

Conforme tela inicial do Clapy na Figura 15 há 9 possibilidades de ações que o usuário pode executar ao clicar nos itens indicados pelas setas, há muitas opções visto que o espaço de tela é limitado (praticamente toda interface é preenchida). Além disso, a tarefa principal, converter o elemento de interface para código exige apenas 2 ações, poucas em relação ao número de possibilidades o que aumenta a carga de trabalho e a probabilidade de ocorrência de erros (CYBIS; BETIOL; FAUST, 2010).

Nesse caso, há 7 itens que podem ser considerados irrelevantes para o processo de uso do plugin, vale ressaltar também a disposição e a separação dos itens da tela e o espaçamento em branco entre os mesmos. Na tela, não há um agrupamento definido, somente um agrupamento na parte inferior com itens semelhantes referentes as opções de ajuda para a aplicação.

Figura 15 – Tela inicial plugin Clapy, itens passíveis de ações pelo usuário



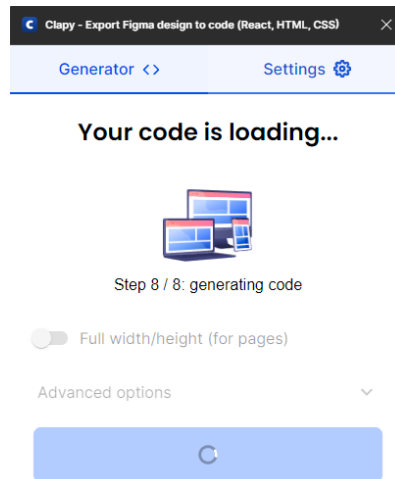
Fonte: Clapy (2023)

Além disso, é notável que há itens que podem ser melhor dispostos em tela, como exemplo cito a seção opções avançadas (*Advanced options*) que poderia ser movida para uma outra aba como *Settings* e as opções de ajuda e documentação que poderiam formar uma nova aba.

6.1.1.3 Feedback Imediato

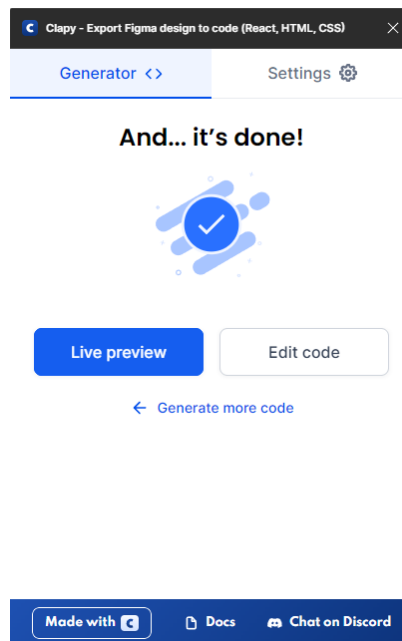
Em relação ao feedback imediato de ações e erros no plugin pode-se classificá-lo como satisfatório, as interfaces indicam o carregamento e execução da tarefa principal textualmente, visualmente através de ícone circular giratório e até indica a progressão da execução através de passos (*steps*) (Figura 16 e Figura 17).

Figura 16 – Tela de processamento Clapy



Fonte: Clapy (2023)

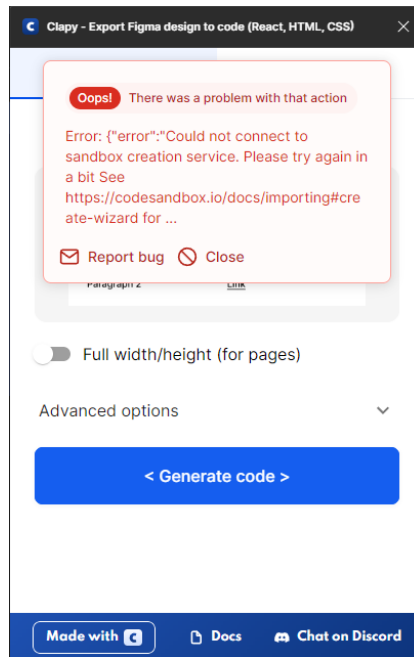
Figura 17 – Tela de conclusão de processamento Clapy



Fonte: Clapy (2023)

Além disso, vale ressaltar que durante a utilização do plugin ocorreu um erro e seu feedback foi dado através de um *pop-up* com sua descrição e a possibilidade de reportá-lo, visualizável na Figura 18.

Figura 18 – Feedback de erro



Fonte: Clapy (2023)

6.1.1.4 Ações mínimas

Para utilização desse plugin há somente 3 ações a serem tomadas, tendo como entrada um elemento ou conjunto de elementos de interface definidos no Figma basta:

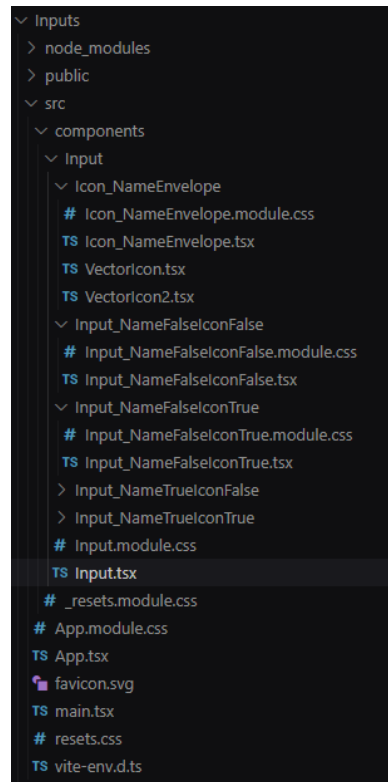
1. Selecionar o elemento ou conjunto de elementos.
2. Clicar em “*Generate code*” e aguardar o processamento da ação.
3. Clicar em “*Edit code*” para fazer o download do código gerado ou abri-lo em uma guia de navegador em um ambiente de desenvolvimento online.

6.1.2 Do resultado final gerado:

6.1.2.1 Nomenclatura

Conforme Figura 19, para conversão dos elementos de interface campos de texto e suas variações o plugin seguiu a nomenclatura utilizada nos arquivos Figma separando código CSS do código em Javascript. O padrão de nomenclatura é formado pela concatenação do nome do elemento, primeira variável e suas variações de valor e, segunda variável e suas variações de valor, e deixa evidente que os arquivos gerados são proporcionais as combinações 2x2 das variáveis.

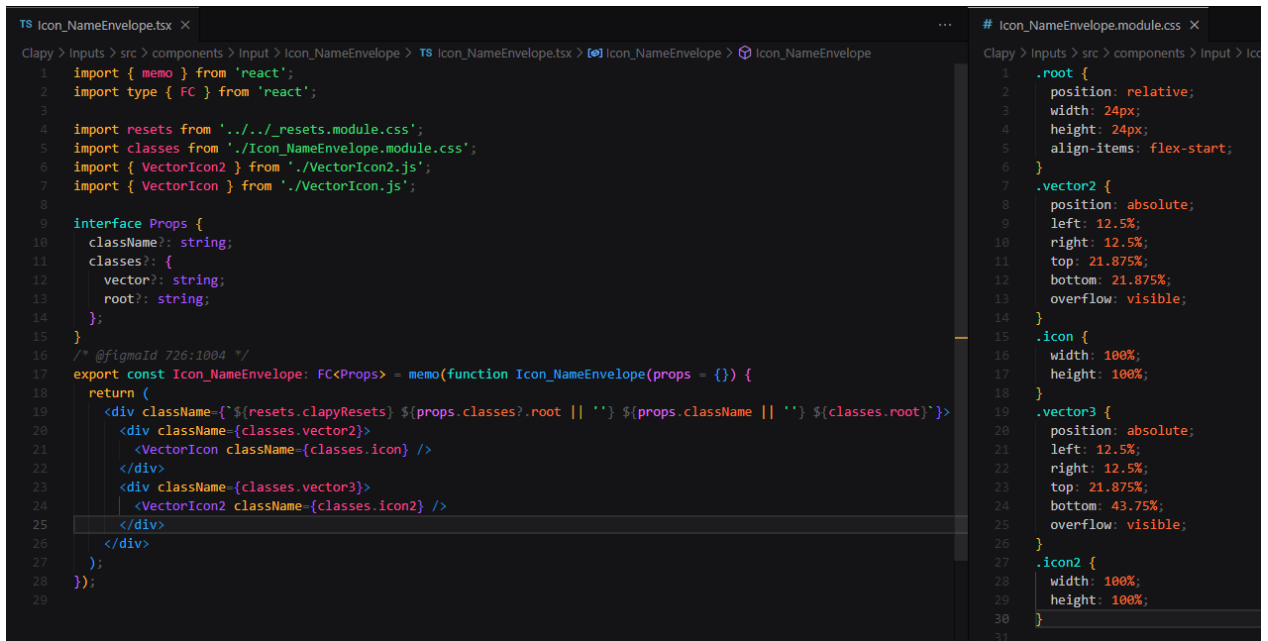
Figura 19 – Arquivos do código gerado com as variações de campos de texto



No geral essa nomeação de arquivos pode ser considerada imprópria visto que não segue os padrões conhecidos de boas práticas de nomenclatura, utilizando padrão único *camelCase* ou *PascalCase* conforme orientações sugeridas por Mozilla (2023d, tradução nossa).

Em relação ao código fonte gerado observado na Figura 20 é visível que este segue o mesmo padrão de nomenclatura dos arquivos baseado nos nomes dos elementos de interface do arquivo Figma dado de entrada, a definição de nome para os componentes se dá utilizando padrão *camelCase* (MOZILLA, 2023d, tradução nossa).

Figura 20 – Clapy, código fonte gerado para campos de texto



```
TS Icon_NameEnvelope.tsx X
Clapy > Inputs > src > components > Input > Icon_NameEnvelope > TS Icon_NameEnvelope.tsx > Icon_NameEnvelope > Icon_NameEnvelope
1 import { memo } from 'react';
2 import type { FC } from 'react';
3
4 import resets from '../_resets.module.css';
5 import classes from './Icon_NameEnvelope.module.css';
6 import { VectorIcon2 } from './VectorIcon2.js';
7 import { VectorIcon } from './VectorIcon.js';
8
9 interface Props {
10   className?: string;
11   classes?: {
12     vector?: string;
13     root?: string;
14   };
15 }
16 /* @figmaId 726:1004 */
17 export const Icon_NameEnvelope: FC<Props> = memo(function Icon_NameEnvelope(props = {}) {
18   return (
19     <div className={` ${resets.clapyResets} ${props.classes?.root || ''} ${props.className || ''} ${classes.root}`>
20       <div className={classes.vector2}>
21         <VectorIcon className={classes.icon} />
22       </div>
23       <div className={classes.vector3}>
24         <VectorIcon2 className={classes.icon2} />
25       </div>
26     </div>
27   );
28 });
29
# Icon_NameEnvelope.module.css X
Clapy > Inputs > src > components > Input > Icon
1
2   position: relative;
3   width: 24px;
4   height: 24px;
5   align-items: flex-start;
6 }
7 .vector2 {
8   position: absolute;
9   left: 12.5%;
10  right: 12.5%;
11  top: 21.875%;
12  bottom: 21.875%;
13  overflow: visible;
14 }
15 .icon {
16   width: 100%;
17   height: 100%;
18 }
19 .vector3 {
20   position: absolute;
21   left: 12.5%;
22   right: 12.5%;
23   top: 21.875%;
24   bottom: 43.75%;
25   overflow: visible;
26 }
27 .icon2 {
28   width: 100%;
29   height: 100%;
30 }
31
```

6.1.2.2 Linhas de código fonte (SLOC)

Métrica é medida para 4 casos de elementos de interface definidas neste trabalho, são apresentados os valores referentes a quantidade de linhas de código CSS e Javascript geradas contabilizadas pelo software Sonarcube:

- Campos de texto e suas variações (Figura 9): 452 linhas.
- Botões de ação e suas variações (Figura 10): 2096 linhas.
- Formulário de login (Figura 13): 999 linhas.
- Formulário de cadastro (Figura 12): 1061 linhas.

6.1.2.3 Número total de módulos, classes e funções:

Métrica medida para 4 casos de interface definidas neste trabalho, são apresentados os valores referentes ao número de regras CSS e número de funções Javascript declaradas no código gerado:

- Campos de texto: 32 regras CSS totais, 9 funções Javascript totais.
- Botões de ação: 105 regras CSS totais, 58 funções Javascript totais.
- Formulário de login (Figura 13): 64 regras CSS totais, 24 funções Javascript totais.
- Formulário de cadastro (Figura 12): 75 regras CSS totais, 17 funções Javascript totais.

6.1.2.4 Critério de reuso

Utilizando como parâmetro de entrada os campos de texto o plugin não reaproveitou as definições de estilo dos elementos, criando classes CSS com definições iguais e repetidas para cada um, mesmo estes sendo definidos como variações de um elemento. O exemplo na Figura 21 abaixo evidencia as repetições de código CSS e corresponde as variações de campo de texto sem ícone (1) e com ícone (2), há repetição de atributos CSS para os elementos filhos de "label", "value" e "frame2" que identificam o rótulo, valor e a caixa visual para inserção de texto, respectivamente.

Figura 21 – Código CSS gerado, campos de texto sem e com ícone

```

# Input_NameTrueIconFalse.module.css
6 height: 79px;
7 align-items: flex-start;
8 }
9 .frame2 {
10 position: absolute;
11 left: 0;
12 right: 0;
13 top: 39.2405%;
14 bottom: 0;
15 align-items: center;
16 padding: 12px 16px;
17 outline: solid 1px #808080;
18 outline-offset: -1px;
19 border-radius: 4px;
20 background-color: #fff;
21 }
22 .value {
23 color: #808080;
24 font-size: 16px;
25 font-weight: 600;
26 font-family: Inter, system-ui, -apple-system, 'Segoe
27 UI', Roboto, 'Helvetica Neue', Arial, 'Noto Sans',
28 'Liberation Sans', sans-serif;
29 z-index: 0;
30 width: min-content;
31 height: min-content;
32 white-space: nowrap;
33 flex-direction: column;
34 }
35 .label {
36 color: #595959;
37 font-size: 16px;
38 font-weight: 600;
39 font-family: Inter, system-ui, -apple-system, 'Segoe
40 UI', Roboto, 'Helvetica Neue', Arial, 'Noto Sans',
41 'Liberation Sans', sans-serif;
42 position: absolute;
43 left: 0;
44 right: 88.25%;
45 top: 0;
46 bottom: 75.9494%;
47 width: min-content;
48 height: min-content;
49 white-space: nowrap;
50 flex-direction: column;
51 }
52 }

# Input_NameTrueIconTrue.module.css
17 height: 79px;
18 align-items: flex-start;
19 }
20 .frame2 {
21 position: absolute;
22 left: 0;
23 right: 0;
24 top: 39.2405%;
25 bottom: 0;
26 align-items: center;
27 gap: 12px;
28 padding: 12px 16px;
29 outline: solid 1px #808080;
30 outline-offset: -1px;
31 border-radius: 4px;
32 background-color: #fff;
33 }
34 .value {
35 color: #808080;
36 font-size: 16px;
37 font-weight: 600;
38 font-family: Inter, system-ui, -apple-system, 'Segoe
39 UI', Roboto, 'Helvetica Neue', Arial, 'Noto Sans',
40 'Liberation Sans', sans-serif;
41 z-index: 0;
42 width: min-content;
43 height: min-content;
44 white-space: nowrap;
45 flex-direction: column;
46 }
47 .label {
48 color: #595959;
49 font-size: 16px;
50 font-weight: 600;
51 font-family: Inter, system-ui, -apple-system, 'Segoe
52 UI', Roboto, 'Helvetica Neue', Arial, 'Noto Sans',
53 'Liberation Sans', sans-serif;
54 position: absolute;
55 left: 0;
56 right: 88.25%;
57 top: 0;
58 bottom: 75.9494%;
59 width: min-content;
60 height: min-content;
61 white-space: nowrap;
62 flex-direction: column;
63 }
64 }

```

Fonte: Clapy (2023)

O mesmo comportamento é observado no código Javascript gerado para os mesmos elementos, conforme Figura 19 abaixo, o único trecho que não é gerado de forma repetida é o referente a importação e referência do ícone (linhas 4 e 14 do código 1 na figura) que é justamente o atributo ou característica que varia nessas variações.

Figura 22 – Código Javascript gerado, campos de texto sem e com ícone

```
TS Input_NameTrueIconFalse.tsx X
Clapy > Inputs > src > components > Input > Input_NameTrueIconFalse > TS Input_NameTrueIconFalse.tsx > ...
1 import { memo } from 'react';
2 import type { FC } from 'react';
3 import resets from '../_resets.module.css';
4 import classes from './Input_NameTrueIconFalse.module.css';
5 interface Props {
6   className?: string;
7 }
8 /* @figmaId 726:1028 */
9 export const Input_NameTrueIconFalse: FC<Props> = memo(function Input_NameTrueIconFalse(props = {}) {
10   return (
11     <div className={` ${resets.clapyResets} ${classes.root}` >
12       <div className={classes.frame2}>
13         <div className={classes.value}>value</div>
14       </div>
15       <div className={classes.label}>Label:</div>
16     </div>
17   );
18 });
19

TS Input_NameTrueIconTrue.tsx X
Clapy > Inputs > src > components > Input > Input_NameTrueIconTrue > TS Input_NameTrueIconTrue.tsx > ...
1 import { memo } from 'react';
2 import type { FC } from 'react';
3 import resets from '../_resets.module.css';
4 import { Icon_NameEnvelope } from '../Icon_NameEnvelope/Icon_NameEnvelope.js';
5 import classes from './Input_NameTrueIconTrue.module.css';
6 interface Props {
7   className?: string;
8 }
9 /* @figmaId 726:927 */
10 export const Input_NameTrueIconTrue: FC<Props> = memo(function Input_NameTrueIconTrue(props = {}) {
11   return (
12     <div className={` ${resets.clapyResets} ${classes.root}` >
13       <div className={classes.frame2}>
14         <Icon_NameEnvelope className={classes.icon} classes={{ vector: classes.vector }} />
15         <div className={classes.value}>value</div>
16       </div>
17       <div className={classes.label}>Label:</div>
18     </div>
19   );
20 });
21
```

Fonte: Clapy (2023)

Para o conjunto de elementos referentes aos botões de ação há o mesmo comportamento, conforme Figura 23 o código Javascript gerado repete exatamente o mesmo código para elementos com hierarquia igual.

Figura 23 – Código Javascript gerado, duas variações de botões de ação

```

TS Button_SizeLargeIconTrueStateH.tsx X
Clapy > Buttons > src > components > Button > Button_SizeLargeIconTrueStateH > TS Button_SizeLargeIconTrueStateH.tsx > ...
1 import { memo } from 'react';
2 import type { FC } from 'react';
3 import resets from '../_resets.module.css';
4 import { Icon_NamePlay } from '../Icon_NamePlay/Icon_NamePlay.js';
5 import classes from './Button_SizeLargeIconTrueStateH.module.css';
6 import { PolygonIcon } from './PolygonIcon.js';
7 interface Props {
8   className?: string;
9 }
10 /* @figmaId 726:474 */
11 export const Button_SizeLargeIconTrueStateH: FC<Props> = memo(function Button_SizeLargeIconTrueStateH(props = {}) {
12   return (
13     <div className={` ${resets.clapyResets} ${classes.root}`}>
14       <Icon_NamePlay
15         className={classes.icon2}
16         swap={{
17           polygon1: <PolygonIcon className={classes.icon} />,
18         }}
19       />
20       <div className={classes.button}>Button</div>
21     </div>
22   );
23 });
24

TS Button_SizeLargeIconTrueStateH2.tsx X
Clapy > Buttons > src > components > Button > Button_SizeLargeIconTrueStateH2 > TS Button_SizeLargeIconTrueStateH2.tsx > Props > className
1 import { memo } from 'react';
2 import type { FC } from 'react';
3 import resets from '../_resets.module.css';
4 import { Icon_NamePlay } from '../Icon_NamePlay/Icon_NamePlay.js';
5 import classes from './Button_SizeLargeIconTrueStateH2.module.css';
6 import { PolygonIcon } from './PolygonIcon.js';
7 interface Props {
8   className?: string;
9 }
10 /* @figmaId 736:1728 */
11 export const Button_SizeLargeIconTrueStateH2: FC<Props> = memo(function Button_SizeLargeIconTrueStateH2(props = {}) {
12   return (
13     <div className={` ${resets.clapyResets} ${classes.root}`}>
14       <Icon_NamePlay
15         className={classes.icon2}
16         swap={{
17           polygon1: <PolygonIcon className={classes.icon} />,
18         }}
19       />
20       <div className={classes.button}>Button</div>
21     </div>
22   );
23 });
24

```

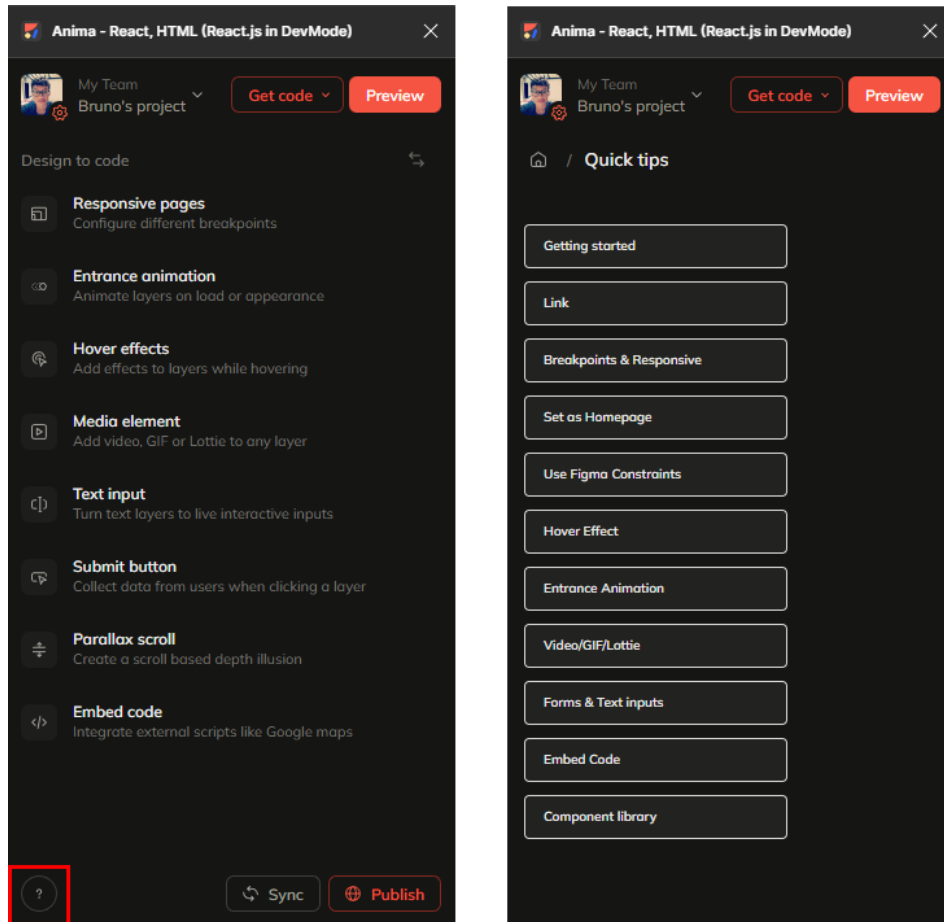
Fonte: Clapy (2023)

6.2 Anima

6.2.1 Da usabilidade:

6.2.1.1 Opções de ajuda e documentação

Figura 24 – Tela inicial plugin Anima, opções de ajuda e documentação



Fonte: Anima (2023)

Conforme janela da interface inicial do Anima (Figura 24) a esquerda há somente 1 ícone que indica acesso a ajuda (destacado em vermelho) na forma de dicas rápidas, ao clicá-lo o plugin abre um menu com várias opções de ajuda e acesso a documentação, cada opção ao ser selecionada apresenta um vídeo com respectivo conteúdo:

- Instruções iniciais (*Get started*): possui passo a passo de utilização da aplicação, descrevendo várias possíveis funcionalidades.
- Instruções de melhores práticas pra montagem de interface no Figma, como por exemplo, como utilizar *breakpoints* e responsividade, efeitos de estado como *hover*, animações, etc.
- Informações em relação a montagem de uma biblioteca de componentes e até como incorporar código externo ao código fonte gerado via próprio plugin.

O plugin não apresenta acesso direto a documentação apesar desta existir de forma extensa e facilmente encontrada em uma busca na web, inclusive com artigos desenvolvidos por desenvolvedores e designers. Porém, segundo o critério de Opções de ajuda e documentação (4.1.1) o acesso a essas informações não é facilitado e ideal, há somente o acesso a instruções

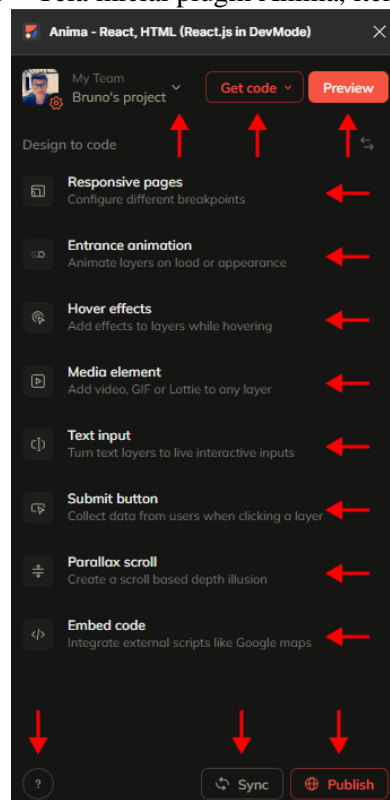
básicas na interface e difícil acesso a documentação completa e possíveis informações em relação a tratamentos de erros ou situações inesperadas, assim, classificando o plugin com opções de ajuda e documentação parciais.

6.2.1.2 Densidade informacional

Em relação a densidade de informações conforme Figura 25 há 14 itens na interface que permitem uma ação do usuário, porém, somente 2 deles de fato fazem a geração de código fonte, os botões “*Get code*” e “*Preview*”. Há uma lista de opções para personalização do código fonte gerado, que permite indicar para o plugin o tipo de elemento de interface a ser convertido, como páginas responsivas, caixas de texto, botões e etc., funções úteis para conversão de elementos individuais de tela de forma otimizada.

Para o plugin em questão o número de itens irrelevantes é 12, porém, vale ressaltar que a tela é melhor organizada no ponto de vista de disposição de elementos, há o agrupamento de itens com a funcionalidade semelhante, sugerir para o plugin um tipo de entrada possível. Além disso, os botões de ações principais encontram-se agrupados na área superior direita e similares, em concordância com as leis de Gestalt de similaridade e de proximidade (GOMES FILHO, 2008).

Figura 25 – Tela inicial plugin Anima, itens clicáveis

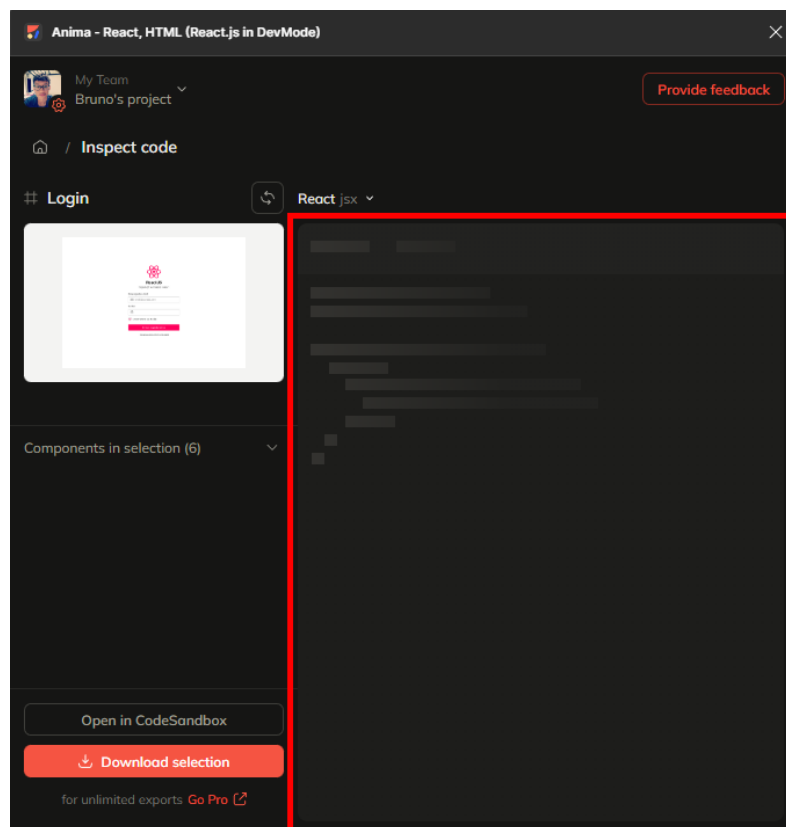


Fonte: Anima (2023)

6.2.1.3 Feedback Imediato

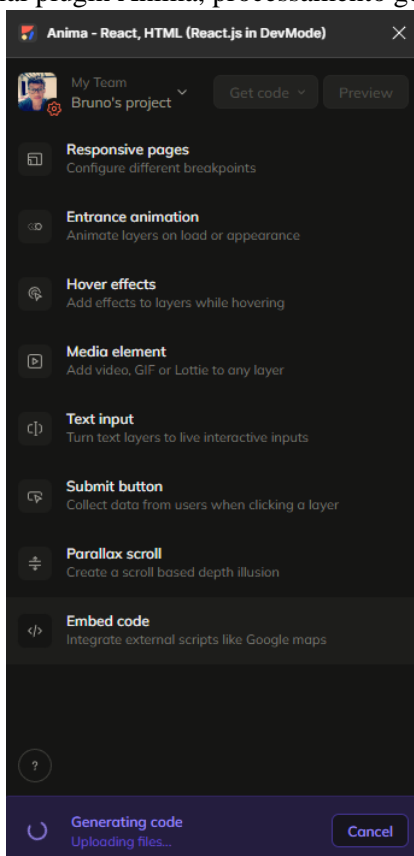
O plugin Anima indica de forma intuitiva alguns processamentos, via texto e animações como observadas nas telas ao realizar a ação de geração de código fonte (Figura 26 e Figura 27), deixando claro para quem utiliza o processo que está sendo executado. Além disso, vale ressaltar que na interface principal o feedback para o usuário se dá sempre na mesma localidade da interface (parte inferior) e de cor destacada, no processamento e na conclusão de execuções.

Figura 26 – Tela inicial plugin Anima, processamento geração de código fonte



Fonte: Anima (2023)

Figura 27 – Tela inicial plugin Anima, processamento geração de código fonte



Fonte: Anima (2023)

Conforme critério de feedback imediato (seção 4.1.3) o plugin pode ser classificado como satisfatório, visto que possui indicações de processamento claras e feedback adequado de conclusão desses processamentos.

6.2.1.4 Ações mínimas

Para geração de código fonte a partir de elementos de interface o plugin exige apenas 3 ações:

1. Selecionar o elemento ou conjunto de elementos de interface;
2. Clicar em “*Get code*”;
3. Clicar nas opções de exportação ou inspeção (submenu com opções “*Inspect*” e “*Export*”)

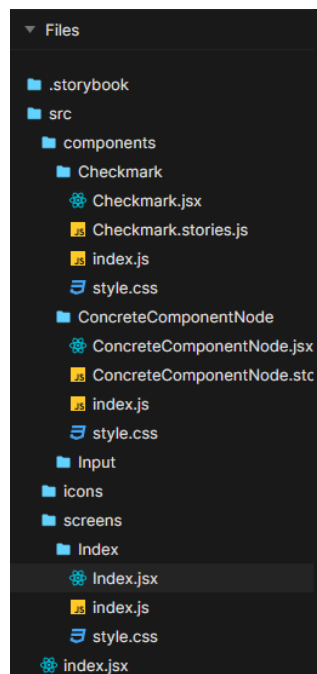
Nesse ponto, vale ressaltar que diferentemente do plugin avaliado na seção anterior (6.1) não há instrução para a primeira ação, em nenhum momento aplicação indica para usuário selecionar algum elemento, o que pode dificultar o primeiro uso de usuários iniciantes.

6.2.2 Do resultado final gerado:

6.2.2.1 Nomenclatura

Ao visualizar a hierarquia de arquivos (Figura 28) fica clara que a nomeação segue um padrão adequado predominantemente *PascalCase* para definição dos arquivos correspondentes aos elementos de interface, é mais limpa em comparação com o plugin avaliado anteriormente. Além disso, é notável que para a mesma entrada houve a geração de um número menor de arquivos, indicando algumas otimizações da conversão.

Figura 28 – Anima arquivos do código gerado com as variações da caixa de seleção



Em relação a nomeação de funções, variáveis e propriedades o código gerado é menor e mais sucinto, sem concatenações extensas de nomes e em *lowercase*, o que torna o código mais agradável aos olhos.

Figura 29 – Anima, código fonte Javascript e CSS gerado para campos de texto

The image shows a code editor with three panels. The top-left panel shows the `Box.jsx` file with a function `Box` that returns a JSX element containing three `Input` components with different class names and props. The bottom-left panel shows the `Input.jsx` file with a function `Input` that takes props like `value`, `label`, `name`, `icon`, and `className`, and returns a JSX element with a `frame` containing an `icon`, a `text-wrapper` with the `value`, and a `label`. The right panel shows the `style.css` file with CSS rules for `.input`, `.input.frame`, `.input.icon`, `.input.value`, and `.input.text-wrapper`.

```

Box.jsx
Anima > AnimaPackage-React-VWupB > src > screens > Box > Box.jsx > ...
4
5 export const Box = () => {
6   return (
7     <div className="box">
8       <div className="input-2">
9         <Input className="name-true-icon-true" icon label="Label:" name />
10        <Input className="name-true-icon-false" icon={false} label="Label:" name value="value" />
11        <Input className="name-false-icon-true" icon name={false} value="value" />
12        <Input className="name-false-icon" icon={false} name={false} value="value" />
13      </div>
14    </div>
15  );
16 };
17

Input.jsx
Anima > AnimaPackage-React-VWupB > src > components > Input > Input.jsx > Input
11 export const Input = ({ value = "value", label = "Label:", name, icon, className }) => {
12   return (
13     <div className={`input name-${name} ${className}`} >
14       {name && (
15         <>
16           <div className="frame">
17             {icon && (
18               <>
19                 <NameEnvelope className="icon" />
20                 <input className="value" />
21               </>
22             )}
23             {icon && <div className="text-wrapper">{value}</div>}
24           </div>
25           <div className="label">{label}</div>
26         </>
27       )}
28     );
29     {!name && icon && <NameEnvelope className="icon" />}
30     {!name && <div className="div">{value}</div>}
31   </div>
32 );
33 };
34 };
35 };
36 };

# style.css
Anima > AnimaPackage-React-VWupB > src > components > Input > # style.css > ...
1 .input {
2   position: relative;
3   width: 400px;
4 }
5
6 .input.frame {
7   align-items: center;
8   background-color: var(--shade-10);
9   border: 1px solid;
10  border-color: var(--neutral-30);
11  border-radius: 4px;
12  display: flex;
13  gap: 12px;
14  height: 48px;
15  left: 0;
16  padding: 12px 16px;
17  position: absolute;
18  top: 31px;
19  width: 400px;
20 }
21
22 .input.icon {
23   height: 24px !important;
24   position: relative !important;
25   width: 24px !important;
26 }
27
28 .input.value {
29   background-color: transparent;
30   border: 0;
31   color: var(--neutral-30);
32   flex: 0 0 auto;
33   font-family: "Inter", Helvetica;
34   font-size: 16px;
35   font-weight: 600;
36   letter-spacing: 0;
37   line-height: normal;
38   padding: 0;
39   position: relative;
40   white-space: nowrap;
41 }
42
43 .input.text-wrapper {
44   color: var(--neutral-30);
45   font-family: "Inter", Helvetica;
46   font-size: 16px;
47   font-weight: 600;
48   letter-spacing: 0;

```

6.2.2.2 Linhas de código fonte (SLOC)

Métrica é medida para 4 casos de elementos de interface definidas neste trabalho, são apresentados os valores referentes a quantidade de linhas de código CSS e Javascript geradas:

- Campos de texto: 459 linhas.
- Botões de ação: 833 linhas.
- Formulário de login (Figura 12): 1004 linhas.
- Formulário de cadastro (Figura 11): 821 linhas.

6.2.2.3 Número total de módulos, classes e funções:

Métrica medida para 4 casos de interface definidas neste trabalho, são apresentados os valores referentes ao número de regras CSS e número de funções Javascript declaradas no código gerado:

- Campos de texto: 16 ou 20 regras CSS totais, 6 funções Javascript totais.

- Botões de ação: 37 ou 73* regras CSS totais, 7 funções Javascript totais.
- Formulário de login (Figura 12): 62 ou 63 regras CSS totais, 14 funções Javascript totais.
- Formulário de cadastro (Figura 11): 56 regras CSS totais, 8 funções Javascript totais.

6.2.2.4 Critério de reuso

Em relação ao reuso de código o plugin apresentou uma geração de código melhor otimizada aproveitando definições de estilo e estrutura, conforme Figura 30, o código Javascript gerado referente ao conjunto de caixas de texto é único para as 4 variações propostas.

Além disso, a única função criada recebe como parâmetros as variáveis que definem as variações para determinar internamente a estrutura conforme o valor dado a elas, recebendo ainda como valores *default* os valores definidos na interface.

Figura 30 – Anima, código fonte Javascript gerado para campos de texto

```

Inputjsx x
Anima > Inputs > src > components > Input > Inputjsx > ...
1 /*
2 We're constantly improving the code you see.
3 Please share your feedback here: https://form.asana.com/?k=uvp-HPqd3\_hyoXRBw1IcNg&d=1152665201300829
4 */
5
6 import PropTypes from "prop-types";
7 import React from "react";
8 import { NameEnvelope } from "../../icons/NameEnvelope";
9 import "./style.css";
10
11 export const Input = ({ value = "value", label = "Label:", name, icon, className }) => {
12   return (
13     <div className={`input name-${name} ${className}`} >
14       {name && (
15         <>
16           <div className="frame">
17             {icon && (
18               <>
19                 <NameEnvelope className="icon" />
20                 <input className="value" />
21               </>
22             )}
23             {!icon && <div className="text-wrapper">{value}</div>}
24           </div>
25           <div className="label">{label}</div>
26         </>
27       )}
28       {!name && icon && <NameEnvelope className="icon" />}
29       {!name && <div className="div">{value}</div>}
30     </div>
31   );
32 };
33
34 Input.propTypes = {
35   value: PropTypes.string,
36   label: PropTypes.string,
37   name: PropTypes.bool,
38   icon: PropTypes.bool,
39 };

```

Fonte: Anima (2023)

O código CSS gerado também foi menor e de acordo com a Figura 31 utilizou uma estratégia para torná-lo dinâmico, nas linhas 81 e 85 a esquerda há seletores que podem e são utilizados no Javascript dinamicamente. Nesse caso, há a concatenação de um valor do tipo *string* com um valor do tipo *boolean* formando os seletores “.input.name-true” e “.input.name-false” que são responsáveis por definir atributos diferentes para um elemento com rótulo e sem rótulo, respectivamente, a depender do valor dado a variável *name*.

Figura 31 – Anima, código fonte CSS gerado para campos de texto

```

# style.css
Anima > Inputs > src > components > input > # style.css > ...
48 letter-spacing: 0;
49 line-height: normal;
50 position: relative;
51 white-space: nowrap;
52 width: fit-content;
53 }
54
55 .input .label {
56 color: var(--neutral-40);
57 font-family: "Inter", Helvetica;
58 font-size: 16px;
59 font-weight: 600;
60 left: 0;
61 letter-spacing: 0;
62 line-height: normal;
63 position: absolute;
64 top: -1px;
65 white-space: nowrap;
66 }
67
68 .input .div {
69 color: var(--neutral-30);
70 font-family: var(--paragraph-p2-regular-font-family);
71 font-size: var(--paragraph-p2-regular-font-size);
72 font-style: var(--paragraph-p2-regular-font-style);
73 font-weight: var(--paragraph-p2-regular-font-weight);
74 letter-spacing: var(--paragraph-p2-regular-letter-spacing);
75 line-height: var(--paragraph-p2-regular-line-height);
76 position: relative;
77 white-space: nowrap;
78 width: fit-content;
79 }
80
81 .input.name-true {
82 height: 79px;
83 }
84
85 .input.name-false {
86 align-items: center;
87 background-color: var(--shade-10);
88 border: 1px solid;
89 border-color: var(--neutral-30);
90 border-radius: 4px;
91 display: flex;
92 gap: 12px;
93 height: 48px;
94 padding: 12px 16px;
95 }
# style.css
Anima > Inputs > src > screens > Box > # style.css > .box
1 .box {
2 height: 557px;
3 position: relative;
4 width: 534px;
5 }
6
7 .box .input-2 {
8 border: 1px dashed;
9 border-color: #9747ff;
10 border-radius: 5px;
11 height: 557px;
12 left: 0;
13 overflow: hidden;
14 position: fixed;
15 top: 0;
16 width: 534px;
17 }
18
19 .box .name-true-icon-true {
20 left: 20px !important;
21 position: absolute !important;
22 top: 20px !important;
23 }
24
25 .box .name-true-icon-false {
26 left: 20px !important;
27 position: absolute !important;
28 top: 116px !important;
29 }
30
31 .box .name-false-icon-true {
32 left: 20px !important;
33 position: absolute !important;
34 top: 225px !important;
35 }
36
37 .box .name-false-icon {
38 left: 20px !important;
39 position: absolute !important;
40 top: 293px !important;
41 }
42

```

Fonte: Anima (2023)

Utilizando como entrada para o plugin o conjunto de botões de ação o plugin manteve o comportamento mais eficaz, conforme a Figura 32, definiu apenas 1 função Javascript representando todas as 36 variações de botões e utilizou os parâmetros dessa função para definir de fato as variações possíveis. Além disso, houve a detecção do tipo de elemento HTML para formação do componente, o próprio plugin definiu um elemento com tags `<button>`.

Figura 32 – Anima, código fonte Javascript gerado para botões de ação

```
Button.jsx x
Anima > Buttons > src > components > Button > Button.jsx > Button
1 /*
2 We're constantly improving the code you see.
3 Please share your feedback here: https://form.asana.com/?k=uvp-HPgd3\_hyoXRBw1IcNg&d=1152665201300829
4 */
5
6 import PropTypes from "prop-types";
7 import React from "react";
8 import { Icon } from "../Icon";
9 import "./style.css";
10
11 export const Button = ({ text = "Button", size, icon, state, type, className }) => {
12   return (
13     <button className={`button ${size} ${state} ${type} ${className}`}>
14       {icon && (
15         <Icon
16           name="play"
17           namePlay={
18             size === "small" && (state === "hover" || type === "primary") && ["secondary", "primary"].includes(type)
19               ? "/img/icon-10.svg"
20             : state === "disabled" && type === "secondary" && ["large", "medium"].includes(size)
21               ? "/img/icon-12.svg"
22             : state === "disabled" && type === "secondary" && size === "small"
23               ? "/img/icon-8.svg"
24             : state === "default" && type === "secondary" && ["large", "medium"].includes(size)
25               ? "/img/icon-4.svg"
26             : state === "default" && type === "secondary" && size === "small"
27               ? "/img/icon-2.svg"
28             : "/img/icon-14.svg"
29           }
30           namePlayClassName={` ${size === "small" ? "class" : "class-2"}`}
31         />
32       )}
33     </button>
34     <div className="text-wrapper">{text}</div>
35   </button>
36 );
37 };
38
39 Button.propTypes = {
40   text: PropTypes.string,
41   size: PropTypes.oneOf(["large", "medium", "small"]),
42   icon: PropTypes.bool,
43   state: PropTypes.oneOf(["default", "disabled", "hover"]),
44   type: PropTypes.oneOf(["primary", "secondary"]),
45 };
```

Fonte: Anima (2023)

Para as definições de CSS geradas evidentes na Figura 33, o plugin previu otimizações e reaproveitou atributos CSS definindo classes que podem ser compartilhadas em mais de uma variação de botão. No caso, há classes para definições de atributos estruturais referentes ao tamanho do botão, como altura, padding e largura de borda, e também para definições variadas de atributos de estilo referentes aos estados do elemento, como cor de fundo, cor de borda e cor de fonte.

Além disso, o plugin utilizou a funcionalidade presente no CSS referente as definições de variáveis para cores e tipografias utilizadas

Figura 33 – Anima, código fonte CSS gerado para os botões de ação

```

# style.css x # style.css x
Anima > Buttons > src > components > Button > # style.css > .button Anima > Buttons > src > components > Button > # style.css > .button
1 .button { 46
2   all: unset; 47
3   align-items: center; 48   .button.secondary {
4   border-radius: 8px; 49     border: 1px solid;
5   box-sizing: border-box; 50   }
6   display: inline-flex; 51   .button.medium {
7   gap: 7px; 52     min-height: 37px;
8   justify-content: center; 53     padding: 5px 15px;
9   position: relative; 54   }
10 } 55
11 56   .button.default.primary {
12 13   .button.class { 57     background-color: var(--primary-40);
14     left: unset !important; 58   }
15     margin-bottom: -3.5px !important; 59
16     margin-top: -3.5px !important; 60   .button.default.secondary {
17     position: relative !important; 61     border-color: var(--primary-40);
18     top: unset !important; 62   }
19   } 63
20 21   .button.class-2 { 64   .button.disabled.primary {
22     left: unset !important; 65     background-color: var(--primary-10);
23     position: relative !important; 66   }
24     top: unset !important; 67
25   } 68   .button.disabled.secondary {
26 27   .button.text-wrapper { 69     border-color: var(--primary-10);
28     position: relative; 70   }
29     width: fit-content; 71
30   } 72   .button.large .text-wrapper {
31 32   .button.large { 73     font-family: var(--paragraph-p1-regular-font-family);
33     min-height: 47px; 74     font-size: var(--paragraph-p1-regular-font-size);
34     padding: 10px 15px; 75     font-style: var(--paragraph-p1-regular-font-style);
35   } 76     font-weight: var(--paragraph-p1-regular-font-weight);
36 37   .button.hover { 77     letter-spacing: var(--paragraph-p1-regular-letter-spacing);
38     background-color: var(--primary-20); 78     line-height: var(--paragraph-p1-regular-line-height);
39     border: 1px solid; 79   }
40     border-color: var(--primary-20); 80
41   } 81   .button.hover .text-wrapper {
42 43   .button.small { 82     color: var(--shade-10);
44     height: 30px; 83   }
45     padding: 5px 10px; 84
46   } 85   .button.small .text-wrapper {
47 48   .button.secondary { 86     font-family: var(--paragraph-p2-regular-font-family);
49     border: 1px solid; 87     font-size: var(--paragraph-p2-regular-font-size);
50   } 88     font-style: var(--paragraph-p2-regular-font-style);
51   } 89     font-weight: var(--paragraph-p2-regular-font-weight);
52   } 90     letter-spacing: var(--paragraph-p2-regular-letter-spacing);
53   } 91     line-height: var(--paragraph-p2-regular-line-height);
54   } 92     margin-top: -0.5px;
55   } 93     white-space: nowrap;
56   } 94
57   } 95
58   } 96
59   } 97
60   } 98
61   } 99
62   } 100
63   } 101
64   } 102
65   } 103
66   } 104
67   } 105
68   } 106
69   } 107
70   } 108
71   } 109
72   } 110
73   } 111
74   } 112
75   } 113
76   } 114
77   } 115
78   } 116
79   } 117
80   } 118
81   } 119
82   } 120
83   } 121
84   } 122
85   } 123
86   } 124
87   } 125
88   } 126
89   } 127
90   } 128
91   } 129
92   } 130
93   } 131
94   } 132
95   } 133
96   } 134
97   } 135
98   } 136
99   } 137
100 } 138

```

Fonte: Anima (2023)

7 RESULTADOS OBTIDOS

7.1 Análise de usabilidade

Em relação aos critérios de usabilidade elencados e conforme é observável na Tabela 7.1 ambos plugins apresentaram resultados parecidos, principalmente em relação aos feedbacks de processamentos para o usuário e número de ações mínimas necessárias para execução da geração do código correspondente a um elemento de interface.

Tabela 7.1 – Síntese da análise de critérios de usabilidade

<i>Plugin</i>	<i>Opções de ajuda e documentação</i>	<i>Feedback imediato</i>	<i>Densidade informacional</i>	<i>Ações mínimas</i>
Clapy	Suficientes	Satisfatório	7 itens irrelevantes	3
Anima	Parciais	Satisfatório	12 itens irrelevantes	3

As opções de ajuda e documentação de ambos plugins apresentam riqueza de informações acessíveis, úteis e até avançadas para usuários. Porém, em relação ao acesso facilitado via interface o plugin Clapy se destaca, visto que traz o acesso direto via link para toda documentação e a um meio de comunicação com sua comunidade via Discord, enquanto o Anima dispõe apenas informações na forma de dicas para os usuários, através de vídeos curtos. Vale ainda ressaltar que o plugin Anima possui documentação completa acessível via pesquisa na web e conta inclusive com artigos mais avançados elaborados por desenvolvedores e designers.

Em relação a densidade informacional avaliando de forma quantitativa ambos também apresentaram resultados semelhantes com uma densidade alta de opções de ações que podem ser consideradas irrelevantes para a tarefa proposta pelos plugins. Avaliando de forma qualitativa o plugin Anima se destacou por agrupar elementos de interface com funcionalidades em comum e com características similares, de acordo com as leis de agrupamento e similaridade de Gestalt (GOMES FILHO, 2008), como os botões na área superior direita para geração de código fonte e sua execução em tempo real, e a lista de itens na parte inferior que permite sugerir o tipo de elemento de interface dado como entrada para a aplicação.

7.2 Análise de resultado

Ao visualizar o código gerado por ambos plugins estudados pode-se discutir alguns pontos referentes a organização e qualidade da saída para cada tipo de conjunto de elementos dado como entrada.

7.2.1 Da nomenclatura

A nomeação gerada de forma automática pelos plugins seguiu os mesmos padrões para definições de nomes de arquivos, variáveis Javascript e classes CSS.

O plugin Clapy foi menos eficaz na nomeação de seus itens, arquivos e nomes de funções que seguiram um padrão composto pela concatenação dos nomes utilizados na definição do protótipo dado como entrada, gerando nomes extensos e com mais informações de contexto do que o necessário (MARTIN et al., 2011). Além disso, o código gerado não segue um padrão *case* comum, como *PascalCase* ou *camelCase*, mas sim uma mistura, utilizando inclusive hífen internos aos nomes.

Diferentemente o plugin Anima apresentou um padrão de nomenclatura mais limpo e curto, sem concatenações ou nomes extensos, além de ser predominantemente *PascalCase* e em alguns casos *lowercase* resultando em um código com leitura mais agradável aos olhos e de acordo com padrões do guia para escrita de código Javascript (MOZILLA, 2023d, tradução nossa).

7.2.2 Da análise estática de código

A análise estática foi realizada utilizando ferramenta Sonarqube para quantificações do código Javascript e de forma manual para quantificação de código CSS e visou principalmente evidenciar melhores otimizações para os códigos gerados.

Ambos os plugins receberam 4 entradas diferentes sendo correspondentes aos elementos de interface definidos no capítulo 3, os valores da análise de resultado obtidos para cada entrada podem ser observados nas tabelas 7.2, 7.3 e 7.4, onde as entradas são identificadas como:

- Entrada 1: referente aos campos de texto e suas variações (Figura 9);
- Entrada 2: referente aos botões de ação e suas variações (Figura 10);
- Entrada 3: referente a um formulário de login (Figura 13);

- Entrada 4: referente a um formulário de cadastro (Figura 12);

Conforme tabelas 7.2 e 7.3 fica claro que o plugin Anima é mais eficaz ao gerar um código menor e possivelmente melhor otimizado, principalmente quando a entrada é um conjunto maior de elementos que compartilham atributos iguais ou que possuem muitas variações de componentes como na Entrada 2 referente ao conjunto das 36 variações de botões de ação. Aparentemente o plugin Anima identificou mais eficientemente as variáveis que formam as variações dos elementos diminuindo o código e o tornando mais condicional e dinâmico, esta observação é visível na comparação do número de linhas de código e do número de módulos e funções Javascript geradas para a Entrada 2 pelos 2 plugins, nas tabelas 7.2 e 7.3 respectivamente.

Tabela 7.2 – Linhas de código fonte (SLOC)

<i>Plugin</i>	<i>Entrada 1</i>	<i>Entrada 2</i>	<i>Entrada 3</i>	<i>Entrada 4</i>
Clapy	452	2096	999	1061
Anima	459	833	1004	821

Tabela 7.3 – Número total de módulos e funções Javascript geradas

<i>Plugin</i>	<i>Entrada 1</i>	<i>Entrada 2</i>	<i>Entrada 3</i>	<i>Entrada 4</i>
Clapy	9	58	24	17
Anima	6	7	14	8

Além disso, houve um melhor reaproveitamento de classes CSS pelo plugin Anima, observável na tabela 7.4 e figura 26 onde é discutido o reuso de código.

Uma observação importante a ser destacada é o fato de que o plugin Anima definiu para os conjuntos de entrada classes CSS que não afetam a estrutura interna ou de estilo de cada elemento, mas somente sua posição em tela como eixos X e Y definidos pelos atributos CSS

top e *left*. Essas classes CSS adicionais de posicionamento foram separadas na contagem, justificando 2 valores para cada entrada na avaliação do plugin Anima na tabela 7.4.

Ao desconsiderar as classes para posicionamento dos elementos o reuso de código CSS pelo plugin Anima é facilmente visualizável na tabela 7.4, visto que gerou uma economia na definição de classes de 50% para Entrada 1 e de aproximadamente 35% para a Entrada 2.

Tabela 7.4 – Número total de módulos ou classes CSS geradas

<i>Plugin</i>	<i>Entrada 1</i>	<i>Entrada 2</i>	<i>Entrada 3</i>	<i>Entrada 4</i>
Clapy	32	105	64	75
Anima	16 ou 20*	37 ou 73*	62 ou 63*	56

7.2.3 Do reuso

Em relação a reutilização e economia de código o plugin Anima foi extremamente eficaz em comparação com plugin Clapy, como visto nos itens anteriores há um número muito inferior de linhas de código, funções e classes geradas. Além disso, houve a detecção autônoma do plugin para que não repetisse código CSS para elementos semelhantes e a geração de código condicional baseado nas variáveis definidas no protótipo Figma facilitando até a utilização do elemento no momento da sua implementação e uso.

Vale ainda ressaltar que mesmo sem a indicação clara do tipo de elemento de interface definido no protótipo o plugin Anima foi capaz de identificar um elemento botão mesmo este sendo representado como um elemento formado por uma caixa retangular arredondada preenchida com texto.

8 CONCLUSÃO

O presente trabalho visou analisar 2 plugins da ferramenta de prototipação Figma com o intuito de definir qual é a aplicação ideal dado um contexto de desenvolvimento de design e software de um sistema mais robusto. Além de, elencar alguns pontos importantes em relação a eficiência na geração de um código que pode facilmente atualizado, melhor compreendido e consequentemente manutenível.

Durante os estudos e observações das aplicações ficou claro que ambas são eficazes para a tarefa proposta, porém, a eficiência no uso e na otimização do resultado difere e tem grande impacto no processo de uso dessas ferramentas em grande escala, principalmente quando se trata de um *design system* robusto e de um processo de desenvolvimento mais ágil.

Além disso, os estudos para o trabalho deixaram claro que para o desenvolvimento principalmente de interfaces web é ideal que o desenvolvedor tenha conhecimentos mais específicos e prévios relativos ao processo de implementação de uma interface e de seus componentes. O conhecimento mais abrangente em relação a uma ferramenta de prototipação e suas funcionalidades certamente facilitam e agilizam o trabalho de produzir um bom software.

Outro ponto importante observado no trabalho é o fato de que a produção de interfaces esta com o processo cada vez mais contínuo, tanto a ferramenta de prototipação quanto a tecnologia para implementar o seu código abstraem dos mesmos conceitos, se utilizam de características computacionais, variáveis, condicionais, estruturas de dados que facilitam essa conexão e fluxo de desenvolvimento do software.

Futuramente para continuidade e complementação deste trabalho pode-se elencar mais critérios importantes a serem considerados, como a acessibilidade, que é inclusa e pode ser gerada através do HTML semântico, além de acrescentar a avaliação de outras tecnologias de CSS do mercado. Outro tópico importante seria a análise da inclusão do uso desses plugins discutidos no processo de desenvolvimento de software de forma mais autônoma e integrada, se já existe essa funcionalidade nos plugins existentes, o custo e os benefícios dessa integração.

9 REFERÊNCIAS

ALMEIDA, L. T.; MIRANDA, J. M. **Código Limpo e seu Mapeamento para Métricas de Código Fonte**. São Paulo. 2010. Disponível em: <https://www.ime.usp.br/~cef/mac499-10/monografias/lucianna-joao/arquivos/monografia.pdf>. Acesso em agosto 2023.

ANIMA. Página inicial. Disponível em: <https://www.animaapp.com/dev-mode>. Acesso em julho 2023.

CLAPY. Página Inicial. 2023. Disponível em: <https://clapy.co/>. Acesso em julho 2023.

CYBIS, Walter; BETIOL, Adriana Holtz; FAUST, Richard. **Ergonomia e Usabilidade 2ª edição: Conhecimentos, Métodos e Aplicações**. Novatec Editora, 2010.

ECMA INTERNATIONAL. ECMAScript 2023 Language Specification. 14 ed. Geneva. Ecma International, 2023.

FIGMA, Inc. Figma The modern interface design tool. 2023. Disponível em: <https://www.figma.com/ui-design-tool/>. Acesso em julho 2023.

FIGMA, Inc. Developers API Documentation. 2023. Disponível em: <https://www.figma.com/developers/api>. Acesso em julho 2023.

FIGMA, Inc. Make plugins for the Figma Community. 2023. Disponível em: <https://help.figma.com/hc/en-us/articles/360039958874-Make-plugins-for-the-Figma-Community>. Acesso em julho 2023.

FIGMA, Inc. Create and use variants. 2023. Disponível em: <https://help.figma.com/hc/en-us/articles/360056440594-Create-and-use-variants>. Acesso em julho 2023.

GARRET, J. J. **The Elements of User Experience: User-Centered Design for the Web and Beyond**. 2 ed. New Riders. 2010.

GOMES FILHO, João. Gestalt do objeto: sistema de leitura visual da forma. 8 ed. Rev.. São Paulo. Escrituras Editora. 2008.

INTERACTION DESIGN FOUNDATION. Learn more about Design Systems. 2023. Disponível em: <https://www.interaction-design.org/literature/topics/design-systems>. Acesso em setembro 2023.

KUROSE, J. F.; ROSS, K. W.. **Redes de Computadores e a Internet**: uma abordagem top-down. 5ª ed.. São Paulo: Pearson Education do Brasil, 2010.

LIE, Håkon Wium; BOS, Bert. Chapter 2, CSS. **Cascading Style Sheets, designing for the Web**. 2 ed. Addison Wesley, 1999.

MARTIN, Robert C. et al; Código Limpo: **Habilidades Práticas do Agile Software**. Edição revisada. Rio de Janeiro, RJ. Alta Books. 30p. 2011.

MEIRELLES, P. R. M. **Monitoramento de métricas de código-fonte em projetos de software livre**. 2013. Tese (Doutorado em Ciências) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013. Disponível em: <https://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/publico/tesePauloMeirelles.pdf>. Acesso em agosto 2023.

META PLATFORMS, Inc. ReactJS Documentation: Getting started. 2023. Disponível em: <https://pt-br.legacy.reactjs.org/docs/getting-started.html> . Acesso em julho 2023.

META OPEN SOURCE. ReactJS Documentation. 2023. Disponível em: <https://pt-br.react.dev/>. Acesso em julho 2023.

MOZILLA. **CSS: Cascading Style Sheets**. 2023. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS>. Acesso em julho 2023.

MOZILLA. **CSS selectors**. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors . Acesso em agosto 2023.

MOZILLA. **DOM: Introdução ao DOM**. 2023. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/API/Document_Object_Model/Introduction Acesso em julho 2023.

MOZILLA. **Guidelines for writing JavaScript code examples**. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/MDN/Writing_guidelines/Writing_style_guide/Code_style_guide/JavaScript#see_also. Acesso em julho 2023.

MOZILLA. HTML: **HyperText Markup Language**. 2023. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTML>. Acesso em julho 2023.

MOZILLA. **JavaScript**. 2023. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Acesso em julho 2023.

MOZILLA. **Utilizando propriedades CSS personalizadas (variáveis)**. 2023. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/CSS/Using_CSS_custom_properties. Acesso em agosto 2023.

SAWAYA, Márcia Regina. **Dicionário de Informática e Internet**. São Paulo. Nobel, 1999.