UNIVERSIDADE FEDERAL DO CEARÁ

Code smells em frameworks front-end

Disciplina: Qualidade de Software
Equipe: Daniel Ulisses (554563), Kleyton Ferreira (555426)
Projeto: Ant Design
Fork: https://github.com/Daniel145l/ant-design

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado no uso de as any dentro da função getLocale, no componente MenuItemLabelWithTag.
Esse any era utilizado para evitar erros do TypeScript ao acessar valores de tradução dinamicamente, já que o compilador não conseguia garantir que o texto utilizado existia dentro do objeto de idiomas.

```
64  const MenuItemLabelWithTag: React.FC<MenuItemLabelProps> = (props) => {
65    const { styles } = useStyle();
66    const { before, after, link, title, subtitle, search, tag, className } = props;
67
68    const [locale] = useLocale(locales);
69
70    const getLocale = (name: string) => {
71      return (locale as any)[name.toLowerCase()] ?? name;
72    };
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi fazer com que o TypeScript aceitasse o acesso às traduções sem o uso de any.
Como o valor utilizado para buscar a tradução é uma string que pode variar, o compilador não conseguia identificar automaticamente se esse valor correspondia a uma chave válida do objeto de idiomas, gerando erros durante a tipagem.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

```
64  const MenuItemLabelWithTag: React.FC<MenuItemLabelProps> = (props) => {
65    const { styles } = useStyle();
66    const { before, after, link, title, subtitle, search, tag, className } = props;
67
68    const [locale] = useLocale(locales);
69
70    type LocaleValues = typeof locales[keyof typeof locales];
71
72    const getLocale = (name: string) => {
73      const key = name.toLowerCase() as keyof LocaleValues;
74      return locale[key] ?? name;
75    }
```

Para resolver esse problema, utilizei os próprios recursos do TypeScript para identificar automaticamente a estrutura do objeto de idiomas.
Com isso, foi possível acessar os valores de forma mais segura, sem precisar desativar a verificação de tipos usando any, mantendo o comportamento original do sistema e

deixando o código mais fácil de entender e manter.

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado no uso de as any dentro do componente Tilt
O any era utilizado porque essa propriedade é adicionada dinamicamente por uma biblioteca externa, e o TypeScript não reconhecia essa extensão do elemento HTML.

```
18  const Tilt: React.FC<TiltProps> = ({ options, ...props }) => {
19    const node = useRef<HTMLDivElement>(null);
20    useEffect(() => {
21      if (node.current) {
22        VanillaTilt.init(node.current, {
23          ...defaultTiltOptions,
24          ...options,
25        });
26      }
27      return () => {
28        (node.current as any)?.vanillaTilt.destroy();
29      };
30    }, []);
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi informar ao TypeScript que o elemento HTML utilizado no componente possui uma propriedade adicional criada pela biblioteca vanilla-tilt.
Como essa propriedade não faz parte do tipo padrão HTMLDivElement, o compilador gerava erro ao tentar acessá-la, o que levou inicialmente ao uso de any para evitar o problema.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi criado um tipo que estende o HTMLDivElement, adicionando explicitamente a propriedade vanillaTilt.
Com isso, o useRef passou a utilizar esse novo tipo, permitindo acessar o método destroy() de forma segura e mantendo a verificação de tipos ativa no TypeScript.

```
18  type VanillaTiltElement = HTMLDivElement & {
19    vanillaTilt?: {
20      destroy: () => void;
21    };
22  };
23
24  const Tilt: React.FC<TiltProps> = ({ options, ...props }) => {
25    const node = useRef<HTMLDivElement>(null);
26    useEffect(() => {
27      if (node.current) {
28        VanillaTilt.init(node.current, {
29          ...defaultTiltOptions,
30          ...options,
31        });
32      }
33      return () => {
34        (node.current as VanillaTiltElement)?.vanillaTilt?.destroy();
35      };
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado no acesso à propriedade yuyanMonitor do objeto global window. O código utilizava window as any para permitir esse acesso, pois o TypeScript não reconhecia essa propriedade como parte do objeto window.

```
35      // Report if necessary
36      const { yuyanMonitor } = window as any;
37      yuyanMonitor?.log({
38        code: 11,
39        msg: `Page not found: ${location.href}; Source: ${document.referrer}`,
40      });
41    }, [isZhCN, pathname, router]);
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi fazer com que o TypeScript reconhecesse uma propriedade que é adicionada globalmente pela aplicação ou por uma biblioteca externa.
Como essa propriedade não faz parte da definição padrão do window, o compilador gerava erro, o que levou ao uso de any para contornar o problema.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

A refatoração foi realizada adicionando a definição do objeto global no arquivo custom-typings.d.ts, que centraliza tipagens específicas do projeto. Isso permitiu remover o uso de any e manter a verificação de tipos ativa.

```
35      // Report if necessary
36      const { yuyanMonitor } = window;
37      yuyanMonitor?.log({
38        code: 11,
39        msg: `Page not found: ${location.href}; Source: ${document.referrer}`,
40      });
41    }, [isZhCN, pathname, router]);
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado na definição do tipo Color, onde a propriedade cleared estava definida como any.
Esse any era utilizado para permitir que o TypeScript aceitasse o valor retornado pelo componente ColorPicker quando a cor era removida.

```
10  type Color = Extract<GetProp<ColorPickerProps, 'value'>, string | { cleared: any }>;
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi entender qual era o papel da propriedade cleared e quais valores ela realmente poderia assumir.
Como o tipo original não deixava isso claro, foi necessário analisar o comportamento do componente para definir um tipo mais específico sem alterar o funcionamento da aplicação.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi feita uma análise, uma solução possível seria a criação de um tipo específico para o cleared, mas outra possibilidade seria o uso do tipo boolean (o que foi usado).

```
10  type Color = Extract<GetProp<ColorPickerProps, 'value'>, string | { cleared: boolean }>;
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado na definição do tipo Color, onde a propriedade cleared estava definida como any.
Esse any era utilizado para permitir que o TypeScript aceitasse o valor retornado pelo componente ColorPicker quando a cor era removida.
Esse code smell se apresenta exatamente como o anterior, mas em um arquivo diferente (index.tsx).

```
43  type Color = Extract<GetProp<ColorPickerProps, 'value'>, string | { cleared: any }>;
```

Minhas principais dificuldades na remoção do code smell são:

Por ser o mesmo caso do code smell anterior, não tive dificuldades para remover, pois já

havia feito a análise para identificar o tipo de cleared.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi feita uma análise, uma solução possível seria a criação de um tipo específico para o cleared, mas outra possibilidade seria o uso do tipo boolean (o que foi usado).

```
43  type Color = Extract<GetProp<ColorPickerProps, 'value'>, string | { cleared: boolean }>;
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado na renderização dinâmica de componentes do Ant Design, onde o componente selecionado era tipado como React.ComponentType<any> para permitir a passagem de propriedades sem restrições.

```
8   const Antd: React.FC<AntdProps> = (props) => {
9     const { component, ...restProps } = props;
10    const Component = (all[component] ?? React.Fragment) as React.ComponentType<any>;
11    return <Component {...restProps} />;
12  };
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi lidar com os componentes que possuem conjuntos de propriedades diferentes.
Como o TypeScript não consegue inferir automaticamente quais props cada componente aceita, o uso de any foi adotado inicialmente para evitar erros de compilação.
Esse caso em específico é bem interessante, pois apresenta um desafio legal. Como os componentes podem ter propriedades diferentes, como saber qual tipo usar?

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, o tipo foi substituído por Record<string, unknown>, que permite a passagem dinâmica de propriedades sem desativar completamente a verificação de tipos.
Essa abordagem mantém a flexibilidade do componente e melhora a segurança do código em relação ao uso de any.

```
8   const Antd: React.FC<AntdProps> = (props) => {
9     const { component, ...restProps } = props;
10    const Component = (all[component] ?? React.Fragment) as React.ComponentType<unknown>;
11    return <Component {...restProps} />;
12  };
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado na definição do objeto allIcons, onde foi utilizado any para permitir

o acesso dinâmico aos ícones importados da biblioteca @ant-design/icons.

```
11    const allIcons: { [key: PropertyKey]: any } = AntdIcons;
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi permitir o acesso dinâmico aos ícones sem perder a flexibilidade do código.
Inicialmente, o uso de any parecia necessário para evitar erros do TypeScript ao acessar os ícones pelo nome, mesmo que a biblioteca já fornecesse tipagem adequada.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi utilizado diretamente o tipo inferido do módulo @ant-design/icons, eliminando a necessidade de tipagem manual.
Essa abordagem mantém a tipagem original da biblioteca, melhora a segurança do código e evita o uso de tipos genéricos desnecessários.

```
11    const allIcons: Record<string, React.ComponentType> = AntdIcons;
12    //solução mais simples
13    // const allIcons = AntdIcons;
```

Nesse caso, a solução também poderia ser “const allIcons = AntdIcons;” (definimos como sendo “mais simples” no comentário, por ser mais curto). Porém, por se tratar de uma refatoração, fazia sentido deixar o código mais descritivo e claro para quem fosse vê-lo, por isso optamos pela utilização do código apresentado na linha 11.

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado na definição do objeto allIcons, onde foi utilizado any para permitir o acesso dinâmico aos ícones importados da biblioteca @ant-design/icons.
Esse é exatamente o mesmo caso do anterior, mas ele se manifesta em outro arquivo.

```
23    const allIcons: { [key: string]: any } = AntdIcons;
```

Minhas principais dificuldades na remoção do code smell são:

Como o caso é igual ao apresentado anteriormente, as dificuldades foram reduzidas, restando apenas uma análise para identificar de que se tratava do mesmo caso.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi utilizado diretamente o tipo inferido do módulo @ant-design/icons, eliminando a necessidade de tipagem manual.
Essa abordagem mantém a tipagem original da biblioteca, melhora a segurança do código e evita o uso de tipos genéricos desnecessários.

```
23    const allIcons: Record<string, React.ComponentType> = AntdIcons;
```

UNIVERSIDADE
FEDERAL DO CEARÁ

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado nas funções isGoodBadImg, onde o parâmetro imgMeta estava tipado como any, mesmo possuindo uma estrutura bem definida dentro do componente.

```
24  function isGoodBadImg(imgMeta: any): boolean {
25    return imgMeta.isGood || imgMeta.isBad;
26  }
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi identificar qual era a estrutura real do objeto recebido pelas funções.
Como esse objeto é construído dinamicamente a partir das propriedades das imagens, foi necessário analisar o fluxo do código para garantir que o tipo utilizado representasse corretamente os dados disponíveis. Eu precisei estudar um pouco mais para saber como usar o Partial na resposta da tipagem.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi reutilizada a interface já existente no código (MateType), utilizando o tipo Partial<MateType> para representar objetos que podem conter apenas parte das propriedades.
Essa abordagem mantém a flexibilidade do código, melhora a segurança de tipos e evita o uso de tipos genéricos desnecessários.
Também foi necessário adicionar um !! no return, transformando qualquer retorno em boolean.

```
24  function isGoodBadImg(imgMeta: Partial<MateType>): boolean {
25    return !!(imgMeta.isGood || imgMeta.isBad);
26  }
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Any Type, identificado nas funções isCompareImg, onde o parâmetro imgMeta estava tipado como any, mesmo possuindo uma estrutura bem definida dentro do componente. Esse caso é bem parecido com o anterior.

```
28  function isCompareImg(imgMeta: any): boolean {
29    return isGoodBadImg(imgMeta) || imgMeta.inline;
30  }
```

Minhas principais dificuldades na remoção do code smell são:

A solução se apresenta de maneira similar ao caso anterior. Então as dificuldades foram

semelhantes também. Entender o uso do Partial e transformar o retorno em boolean.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Para remover o uso de any, foi reutilizada a interface já existente no código (MateType), utilizando o tipo Partial<MateType> para representar objetos que podem conter apenas parte das propriedades.
Essa abordagem mantém a flexibilidade do código, melhora a segurança de tipos e evita o uso de tipos genéricos desnecessários.
Também foi necessário adicionar um !! no return, transformando qualquer retorno em boolean.

```
28  function isCompareImg(imgMeta: Partial<MateType>): boolean {
29    return isGoodBadImg(imgMeta) || !!imgMeta.inline;
30  }
```

Non Null Assertions

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Non-Null Assertion, identificado no uso do operador ! ao acessar a propriedade linkPath. Esse operador estava sendo utilizado para forçar o TypeScript a aceitar um valor que pode ser undefined, escondendo um possível erro em tempo de execução.

```
110     return {
111       label: isLink ? (
112         <Link to={getLocalizedPathname(linkPath!, isZhCN(pathname), search)}>
113           <FormattedMessage id={id} />
114         </Link>
115       ) : (
116         <FormattedMessage id={id} />
117       ),
118       icon,
119       key: key || i,
120       extra: showBadge ? (showBadge() ? badge : null) : extra,
121     };
122   });
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi garantir que a propriedade linkPath estivesse realmente definida antes de ser utilizada.
Como essa propriedade é opcional no objeto de configuração, o TypeScript corretamente alertava sobre o risco de acesso inválido, o que levou inicialmente ao uso do operador !.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

A refatoração foi feita adicionando uma verificação explícita da existência de linkPath antes de utilizá-la.
Dessa forma, o código passou a lidar corretamente com valores opcionais, eliminando o uso do operador de non-null assertion e tornando o comportamento mais seguro e previsível.

```
110     return {
111       label:
112        isLink && linkPath ? (
113         <Link to={getLocalizedPathname(linkPath, isZhCN(pathname), search)}>
114           <FormattedMessage id={id} />
115         </Link>
116        ) : (
117         <FormattedMessage id={id} />
118        ),
119        icon,
120        key: key || i,
121        extra: showBadge ? (showBadge() ? badge : null) : extra,
122     };
123   });
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Non-Null Assertion, identificado no uso do operador ! ao inicializar um useRef com valor null.
O operador estava sendo utilizado para forçar o TypeScript a aceitar um valor que, inicialmente, pode ser nulo.

```
38    const timerRef = useRef<ReturnType<typeof setTimeout>>(null!);
```

Minhas principais dificuldades na remoção do code smell são:

Non-Null Assertion, identificado no uso do operador ! ao inicializar um useRef com valor null.
O operador estava sendo utilizado para forçar o TypeScript a aceitar um valor que, inicialmente, pode ser nulo.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

A refatoração foi feita ajustando o tipo do useRef para aceitar null, refletindo corretamente o estado inicial da referência.
Além disso, foi adicionada uma verificação antes de chamar clearTimeout, eliminando a necessidade do operador de non-null assertion.

```
38    const timerRef = useRef<ReturnType<typeof setTimeout> | null>(null);
```

UNIVERSIDADE FEDERAL DO CEARÁ

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Non-Null Assertion, identificado no uso do operador ! ao acessar propriedades como theme, isDark e isMobile.
Esses operadores estavam sendo usados para forçar o TypeScript a aceitar valores que podem ser undefined.

```
222   const siteContextValue = React.useMemo<SiteContextProps>(
223     () => ({
224       direction,
225       updateSiteConfig,
226       theme: theme!,
227       isDark: isDark!,
228       isMobile: isMobile!,
```

Minhas principais dificuldades na remoção do code smell são:

A principal dificuldade foi lidar com o fato de que o estado do site é definido como parcial, permitindo valores indefinidos.
Mesmo com valores padrão aplicados na desestruturação, o TypeScript não consegue garantir que essas propriedades sempre existirão.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

A refatoração foi feita substituindo o uso do operador de non-null assertion por valores padrão explícitos no momento da criação do contexto.
Dessa forma, o código passou a tratar corretamente os casos onde os valores poderiam estar indefinidos, sem ocultar possíveis erros.

```
222   const siteContextValue = React.useMemo<SiteContextProps>(
223     () => ({
224       direction,
225       updateSiteConfig,
226       theme: theme ?? [],
227       isDark: isDark ?? false,
228       isMobile: isMobile ?? false,
229       bannerVisible,
230       dynamicTheme,
231     }),
232     [isMobile, direction, updateSiteConfig, theme, isDark, bannerVisible, dynamicTheme],
233   );
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertions nas linhas 316-317 do `table/InternalTable.tsx`. Identifiquei uso de `!` para forçar acesso a `tblRef.current` e `rootRef.current` ignorando possibilidade de serem null.

```
useProxyImperativeHandle(ref, () => ({
  ...tblRef.current!,
  nativeElement: rootRef.current!,
}));
```

Minhas principais dificuldades na remoção do code smell são: garantir que o imperativeHandle não exponha objeto inválido caso as refs não estejam disponíveis durante o ciclo de vida inicial do componente.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Type Guard com verificação explícita `if (!tblRef.current || !rootRef.current)`, retornando objeto vazio tipado em casos edge case para evitar crashes.

```
useProxyImperativeHandle(ref, () => {
if (!tblRef.current || !rootRef.current) {
    return {} as RcReference & { nativeElement: HTMLDivElement };
  }
  return {
    ...tblRef.current,
    nativeElement: rootRef.current,
  };
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertion na linha 364 do `table/InternalTable.tsx`. Identifiquei uso de `!` para forçar existência de `pageSize` ao chamar `onChange` da paginação.

```
if (pagination) {
  pagination.onChange?.(1, changeInfo.pagination?.pageSize!);
}
```

Minhas principais dificuldades na remoção do code smell são: verificar se pageSize existe antes de passar para o callback, evitando passar undefined.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Type Guard adicionando verificação `changeInfo.pagination?.pageSize` na condição do

if, eliminando necessidade da Non Null Assertion.

```
if (pagination && changeInfo.pagination?.pageSize) {
  pagination.onChange?.(1, changeInfo.pagination.pageSize);
}
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertion na linha 370 do `table/InternalTable.tsx`. Identifiquei uso de `!` dentro de arrow function do `getContainer`, apesar da verificação externa já garantir existência do elemento.

```
if (scroll && scroll.scrollToFirstRowOnChange !== false && internalRefs.body.current) {
  scrollTo(0, {
    getContainer: () => internalRefs.body.current!,
  });
}
```

Minhas principais dificuldades na remoção do code smell são: TypeScript não consegue inferir que a ref continua válida dentro da arrow function, mesmo com verificação prévia no if externo.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: captura do valor da ref em variável local antes da arrow function, permitindo que TypeScript mantenha o type narrowing dentro do escopo.

```
if (scroll && scroll.scrollToFirstRowOnChange !== false && internalRefs.body.current) {
  const bodyElement = internalRefs.body.current;
  scrollTo(0, {
    getContainer: () => bodyElement,
  });
}
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertions nas linhas 374-377 do `table/InternalTable.tsx`. Identifiquei múltiplos usos de `!` para forçar propriedades do `changeInfo` como não-null ao chamar onChange.

```
onChange?.(changeInfo.pagination!, changeInfo.filters!, changeInfo.sorter!, {
  currentDataSource: getFilterData(
    getSortData(rawData, changeInfo.sorterStates!, childrenColumnName),
    changeInfo.filterStates!,
    childrenColumnName,
  ),
  action,
});
```

Minhas principais dificuldades na remoção do code smell são: garantir que todas as propriedades necessárias estejam definidas antes de chamar onChange, pois o callback espera valores não-null.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Type Guard composto verificando todas as propriedades necessárias (`pagination`, `filters`, `sorter`, `sorterStates`, `filterStates`) antes de executar o callback.

```
if (changeInfo.pagination && changeInfo.filters && changeInfo.sorter &&
    changeInfo.sorterStates && changeInfo.filterStates) {
  onChange?.(changeInfo.pagination, changeInfo.filters, changeInfo.sorter, {
    currentDataSource: getFilterData(
      getSortData(rawData, changeInfo.sorterStates, childrenColumnName),
      changeInfo.filterStates,
      childrenColumnName,
    ),
    action,
  });
}
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertion na linha 489 do `table/InternalTable.tsx`. Identifiquei uso de `!` para forçar `total` como número na comparação de paginação dinâmica.

```
if (mergedData.length < total!) {
```

Minhas principais dificuldades na remoção do code smell são: garantir que a comparação numérica só ocorra quando total realmente é um número, evitando comportamento inesperado.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Type Guard com `typeof total === 'number'`, verificando explicitamente o tipo antes da comparação e eliminando necessidade da Non Null Assertion.

```
if (typeof total === 'number' && mergedData.length < total) {
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertions nas linhas 544 e 549 do `table/InternalTable.tsx`. Identifiquei duplo uso de `!` para forçar `expandIconColumnIndex` como número ao comparar e decrementar.

```
if (expandType === 'nest' && mergedExpandable.expandIconColumnIndex === undefined) {
  mergedExpandable.expandIconColumnIndex = rowSelection ? 1 : 0;
} else if (mergedExpandable.expandIconColumnIndex! > 0 && rowSelection) {
  mergedExpandable.expandIconColumnIndex! -= 1;
}
```

Minhas principais dificuldades na remoção do code smell são: garantir que expandIconColumnIndex seja número antes de operações matemáticas, já que pode ser undefined.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Type Guard com `typeof expandIconColumnIndex === 'number'`, permitindo TypeScript inferir o tipo correto dentro do bloco e eliminando ambas Non Null Assertions.

```
if (expandType === 'nest' && mergedExpandable.expandIconColumnIndex === undefined) {
  mergedExpandable.expandIconColumnIndex = rowSelection ? 1 : 0;
} else if (typeof mergedExpandable.expandIconColumnIndex === 'number' &&
           mergedExpandable.expandIconColumnIndex > 0 &&
           rowSelection) {
  mergedExpandable.expandIconColumnIndex -= 1;
}
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Non Null Assertion na linha 550 do `table/InternalTable.tsx`. Identifiquei uso de `!` para forçar `tableLocale` como não-null ao passar para `renderExpandIcon`.

```
mergedExpandable.expandIcon =
  mergedExpandable.expandIcon || expandIcon || renderExpandIcon(tableLocale!);
```

UNIVERSIDADE FEDERAL DO CEARÁ

Minhas principais dificuldades na remoção do code smell são: garantir que renderExpandIcon receba um objeto válido mesmo quando tableLocale é undefined, mantendo funcionalidade padrão.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: Nullish Coalescing (`tableLocale || {}`), fornecendo objeto vazio como fallback seguro quando tableLocale é undefined.

```
mergedExpandable.expandIcon =
  mergedExpandable.expandIcon || expandIcon || renderExpandIcon(tableLocale || {});
```

UNIVERSIDADE FEDERAL DO CEARÁ

Direct DOM Manipulation

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 57 do `CodeBlockButton.tsx`. Identifiquei uso de `document.getElementById` para verificar se um script externo já foi carregado, quebrando o paradigma declarativo do React.

```
const existScript = document.getElementById(scriptId) as HTMLScriptElement;
```

Minhas principais dificuldades na remoção do code smell são: garantir que a verificação de script carregado funcione corretamente ao substituir por useRef, mantendo compatibilidade com o atributo `dataset.loaded`.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: substituição de `document.getElementById` por `useRef` do React, mantendo referência controlada ao elemento script durante todo o ciclo de vida do componente.

```
const scriptRef = useRef<HTMLScriptElement | null>(null);
if (scriptRef.current?.dataset.loaded) {
  openHituCodeBlockFn();
  return;
}
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 63 do `CodeBlockButton.tsx`. Identifiquei criação manual de elemento `<script>` e inserção direta no DOM via `document.body.appendChild`, ignorando o ciclo de vida do React.

```
const script = document.createElement('script');
script.src = `https://renderoffice.alipayobjects.com/p/yuyan/180020010001206410/parseFileData-v1.0.1.js?t=${Date.now()}`;
script.async = true;
script.id = scriptId;
script.onload = () => {
  script.dataset.loaded = 'true';
  openHituCodeBlockFn();
};
script.onerror = () => {
  openHituCodeBlockFn();
};
document.body.appendChild(script);
```

Minhas principais dificuldades na remoção do code smell são: mover a lógica de carregamento de script para useEffect mantendo comportamento assíncrono e limpeza adequada do script ao desmontar o componente.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: encapsulamento da manipulação DOM dentro de useEffect com função de cleanup, usando useRef para manter referência ao script criado e permitir

```
const loadScript = () => {
    if (scriptRef.current?.dataset.loaded) return;

    const script = document.createElement('script');
    script.src = `https://renderoffice.alipayobjects.com/p/yuyan/180020010001206410/parseFileData-v1.0.1.js?t=${Date.now()}`;
    script.async = true;
    script.id = scriptId;
    script.onload = () => {
      script.dataset.loaded = 'true';
      scriptRef.current = script;
      openHituCodeBlockFn();
    };
    script.onerror = openHituCodeBlockFn;

    document.body.appendChild(script);
    scriptRef.current = script;

    return () => {
      if (scriptRef.current) {
        document.body.removeChild(scriptRef.current);
      }
    };
  };

  return loadScript();
}, []);
```

UNIVERSIDADE FEDERAL DO CEARÁ

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 295 do `CodePreviewer.tsx`. Identifiquei criação de elemento `<style>` e inserção direta no `<head>` via `document.head.appendChild`, além de uso de type assertion `as any` para adicionar atributo.

```
useEffect(() => {
  if (!style) {
    return;
  }
  const styleTag = document.createElement('style');
  styleTag.innerHTML = style;
  (styleTag as any)['data-demo-url'] = demoUrlWithTheme;
  document.head.appendChild(styleTag);
  return () => {
    document.head.removeChild(styleTag);
  };
}, [style, demoUrlWithTheme]);
```

Minhas principais dificuldades na remoção do code smell são: garantir limpeza correta do style tag ao atualizar ou desmontar, evitando acúmulo de tags no head e verificando se o elemento ainda existe antes de remover.

Métodos de refatoração aplicados: Eu estou usando os seguintes métodos de refatoração para remover o code smell: uso de useRef para rastrear o styleTag criado, substituição de type assertion por `setAttribute`, e verificação `document.head.contains()` antes de remover para evitar erros.

```tsx
const styleRef = useRef<HTMLStyleElement | null>(null);

useEffect(() => {
  if (!style) {
    return;
  }

  // Cleanup previous style if exists
  if (styleRef.current) {
    document.head.removeChild(styleRef.current);
  }

  const styleTag = document.createElement('style');
  styleTag.innerHTML = style;
  styleTag.setAttribute('data-demo-url', demoUrlWithTheme);
  styleRef.current = styleTag;
  document.head.appendChild(styleTag);

  return () => {
    if (styleRef.current && document.head.contains(styleRef.current)) {
      document.head.removeChild(styleRef.current);
      styleRef.current = null;
    }
  };
}, [style, demoUrlWithTheme]);
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 54 do `DocLayout/index.tsx`. Identifiquei uso de `document.getElementById` dentro de useEffect para buscar e remover elemento de estilo do nprogress.

```tsx
useEffect(() => {
  const nprogressHiddenStyle = document.getElementById('nprogress-style');
  timerRef.current = setTimeout(() => {
    nprogressHiddenStyle?.remove();
  }, 0);
  return () => clearTimeout(timerRef.current);
}, []);
```

Minhas principais dificuldades na remoção do code smell são: capturar referência ao elemento no momento correto do ciclo de vida, pois ele pode não existir na montagem

inicial do componente.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: separação da captura da referência em useLayoutEffect e uso de useRef para armazenar o elemento, eliminando busca DOM a cada execução.

```
const nprogressStyleRef = useRef<HTMLElement | null>(null);

useLayoutEffect(() => {
  nprogressStyleRef.current = document.getElementById('nprogress-style');
}, []);

useEffect(() => {
  timerRef.current = setTimeout(() => {
    nprogressStyleRef.current?.remove();
    nprogressStyleRef.current = null;
  }, 0);
  return () => clearTimeout(timerRef.current);
}, []);
```

UNIVERSIDADE FEDERAL DO CEARÁ

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 36 do `toolbarRender.tsx`. Identifiquei criação de elemento `<a>` temporário, inserção no DOM via `document.body.appendChild`, e remoção imediata com `link.remove()` para download de imagem.

```
const onDownload = () => {
  const url = imageList[current];
  const suffix = url.slice(url.lastIndexOf('.'));
  const filename = Date.now() + suffix;

  fetch(url)
    .then((response) => response.blob())
    .then((blob) => {
      const blobUrl = URL.createObjectURL(new Blob([blob]));
      const link = document.createElement('a');
      link.href = blobUrl;
      link.download = filename;
      document.body.appendChild(link);
      link.click();
      URL.revokeObjectURL(blobUrl);
      link.remove();
    });
};
```

Minhas principais dificuldades na remoção do code smell são: reutilizar o elemento link entre múltiplos downloads e garantir limpeza adequada ao desmontar o componente, evitando vazamento de memória.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: uso de useRef para criar link uma única vez e reutilizá-lo, com cleanup no useEffect para remover o elemento apenas na desmontagem do componente.

```tsx
const linkRef = useRef<HTMLAnchorElement | null>(null);

const onDownload = () => {
  const url = imageList[current];
  const suffix = url.slice(url.lastIndexOf('.'));
  const filename = Date.now() + suffix;

  fetch(url)
    .then((response) => response.blob())
    .then((blob) => {
      const blobUrl = URL.createObjectURL(new Blob([blob]));

      if (!linkRef.current) {
        linkRef.current = document.createElement('a');
        document.body.appendChild(linkRef.current);
      }

      linkRef.current.href = blobUrl;
      linkRef.current.download = filename;
      linkRef.current.click();

      URL.revokeObjectURL(blobUrl);
    });
};

// Cleanup no useEffect
useEffect(() => {
  return () => {
    if (linkRef.current) {
      document.body.removeChild(linkRef.current);
      linkRef.current = null;
    }
  };
}, []);
```

UNIVERSIDADE
FEDERAL DO CEARÁ

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 274 do `form.test.tsx`. Identifiquei uso de `document.getElementById` em teste para obter referência ao input, ignorando queries do React Testing Library.

```
const inputNode = document.getElementById('test');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Minhas principais dificuldades na remoção do code smell são: adicionar atributo `role` ao input para permitir busca semântica e substituir todas as referências que usavam o id do elemento.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: substituição de `document.getElementById` por `getByRole` do React Testing Library, seguindo boas práticas de testes acessíveis e semânticos.

```
const { getByRole } = render(
  <ConfigProvider form={{ scrollToFirstError: { block: 'center' } }}>
    <Form onFinishFailed={onFinishFailed}>
      <Form.Item name="test" rules={[{ required: true }]}>
        <input role="textbox" />
      </Form.Item>
      <Form.Item>
        <Button htmlType="submit">Submit</Button>
      </Form.Item>
    </Form>
  </ConfigProvider>,
);

const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
  expect(onFinishFailed).toHaveBeenCalled();
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 303 do `form.test.tsx`. Encontrei outro uso de `document.getElementById` em teste, mesmo padrão da linha 274 mas em contexto de teste diferente.

```
const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Minhas principais dificuldades na remoção do code smell são: manter consistência com a refatoração anterior e garantir que o teste continue validando o comportamento de scroll corretamente.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: aplicação do mesmo padrão da refatoração anterior, usando `getByRole` do React Testing Library para busca semântica do input.

```
const { getByRole } = render(
  <ConfigProvider>
    <Form scrollToFirstError={{ block: 'center' }} onFinishFailed={onFinishFailed}>
      <Form.Item name="test" rules={[{ required: true }]}>
        <input role="textbox" />
      </Form.Item>
      <Form.Item>
        <Button htmlType="submit">Submit</Button>
      </Form.Item>
    </Form>
  </ConfigProvider>,
);

const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 423 do `form/__tests__/index.test.tsx`. Identifiquei uso de `document.getElementById('scroll_test')` em teste de scrollToField, ignorando

queries semânticas do React Testing Library.

```
const inputNode = document.getElementById('scroll_test');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'start',
  scrollMode: 'if-needed',
  });
});
```

Minhas principais dificuldades na remoção do code smell são: garantir que o teste continue validando o comportamento de scroll corretamente ao trocar de getElementById para getByRole, mantendo cobertura de testes.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: substituição de `document.getElementById` por `getByRole('textbox')` do React Testing Library, seguindo padrão de testes acessíveis e eliminando acesso direto ao DOM.

```
const { getByRole } = render(<Demo />);

const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'start',
  scrollMode: 'if-needed',
});
```

```
const inputNode = document.getElementById('test');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Minhas principais dificuldades na remoção do code smell são: adicionar atributo role ao input mantendo compatibilidade com o teste existente e garantir que a asserção continue validando o scroll correto.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: aplicação do padrão de `getByRole` do React Testing Library, adicionando `role="textbox"` ao input para busca semântica e eliminando manipulação direta do DOM.

```
  const { container, getByRole } = render(
  <Form scrollToFirstError={{ block: 'center' }} onFinishFailed={onFinishFailed}>
    <Form.Item name="test" rules={[{ required: true }]}>
      <input role="textbox" />
    </Form.Item>
    <Form.Item>
      <Button htmlType="submit">Submit</Button>
    </Form.Item>
  </Form>,
);

expect(scrollIntoView).not.toHaveBeenCalled();
fireEvent.submit(container.querySelector('form')!);
await waitFakeTimer();

const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Direct DOM Manipulation na linha 636 do `form/__tests__/index.test.tsx`. Identifiquei uso de `document.getElementById('test')` em teste de scrollToFirstError com focus,

terceira ocorrência do mesmo padrão.

```
const inputNode = document.getElementById('test');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});
```

Minhas principais dificuldades na remoção do code smell são: manter consistência com refatorações anteriores e garantir que tanto o scroll quanto o focus sejam testados corretamente após a mudança.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: reutilização do padrão estabelecido usando `getByRole('textbox')` do React Testing Library, mantendo consistência entre os 3 testes similares e eliminando acesso direto ao DOM.

```
const { container, getByRole } = render(
  <Form scrollToFirstError={{ block: 'center', focus: true }} onFinishFailed={onFinishFailed}>
    <Form.Item name="test" rules={[{ required: true }]}>
      <input role="textbox" />
    </Form.Item>
    <Form.Item>
      <Button htmlType="submit">Submit</Button>
    </Form.Item>
  </Form>,
);

expect(scrollIntoView).not.toHaveBeenCalled();

fireEvent.submit(container.querySelector('form')!);
await waitFakeTimer();

const inputNode = getByRole('textbox');
expect(scrollIntoView).toHaveBeenCalledWith(inputNode, {
  block: 'center',
  scrollMode: 'if-needed',
});

expect(inputNode).toHaveFocus();
```

UNIVERSIDADE FEDERAL DO CEARÁ

Multiple Booleans For State

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 191-196 do arquivo `Typography/Base/index.tsx`. Encontrei 5 estados booleanos separados controlando ellipsis, dificultando rastrear mudanças.

```
const [isLineClampSupport, setIsLineClampSupport] = React.useState(false);
const [isTextOverflowSupport, setIsTextOverflowSupport] = React.useState(false);

const [isJsEllipsis, setIsJsEllipsis] = React.useState(false);
const [isNativeEllipsis, setIsNativeEllipsis] = React.useState(false);
const [isNativeVisible, setIsNativeVisible] = React.useState(true);
```

Minhas principais dificuldades na remoção do code smell são:

Minhas principais dificuldades na remoção do code smell são: ajustar chamadas de setState espalhadas pelo componente para usar spread operator sem quebrar useEffects.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Eu estou usando os seguintes métodos de refatoração para remover o code smell: agrupamento em objeto tipado usando Type Inference do TypeScript.

```
type EllipsisState = {
  lineClampSupport: boolean;
  textOverflowSupport: boolean;
  jsEllipsis: boolean;
  nativeEllipsis: boolean;
  nativeVisible: boolean;
};

const [ellipsisState, setEllipsisState] = React.useState<EllipsisState>({
  lineClampSupport: false,
  textOverflowSupport: false,
  jsEllipsis: false,
  nativeEllipsis: false,
  nativeVisible: true,
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 80-81 do `component-token.tsx`. Identifiquei 2 estados booleanos (`bordered`, `loading`) controlando estilos da tabela.

```
const [bordered, setBordered] = useState(false);
const [loading, setLoading] = useState(false);
```

Minhas principais dificuldades na remoção do code smell são:

Minhas principais dificuldades na remoção do code smell são: manter compatibilidade com handlers existentes que são passados para componentes Switch.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

Eu estou usando os seguintes métodos de refatoração para remover o code smell: consolidação em objeto tipado com Type Inference, reduzindo useState de 2 para 1.

```
type TableStyleState = {
  bordered: boolean;
  loading: boolean;
};

const [styleState, setStyleState] = useState<TableStyleState>({
  bordered: false,
  loading: false,
});

const handleBorderChange = (enable: boolean) => {
  setStyleState(prev => ({ ...prev, bordered: enable }));
};

const handleLoadingChange = (enable: boolean) => {
  setStyleState(prev => ({ ...prev, loading: enable }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell:

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple

Booleans For State nas linhas 86-88 do `component-token.tsx`. Encontrei 3 estados booleanos controlando visibilidade de seções da tabela.

```
const [showTitle, setShowTitle] = useState(false);
const [showHeader, setShowHeader] = useState(true);
const [showFooter, setShowFooter] = useState(true);
```

Minhas principais dificuldades na remoção do code smell são:

Minhas principais dificuldades na remoção do code smell são: garantir que os valores iniciais diferentes (false para title, true para header/footer) fossem preservados.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: agrupamento em objeto tipado usando Type Inference, facilitando gerenciamento centralizado de visibilidade.

```
type TableSectionVisibility = {
  showTitle: boolean;
  showHeader: boolean;
  showFooter: boolean;
};

const [sectionVisibility, setSectionVisibility] = useState<TableSectionVisibility>({
  showTitle: false,
  showHeader: true,
  showFooter: true,
});

const handleTitleChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showTitle: enable }));
};

const handleHeaderChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showHeader: enable }));
};

const handleFooterChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showFooter: enable }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State na linha 90 do `component-token.tsx`. Identifiquei o estado `hasData` isolado, que poderia ser agrupado com outros estados relacionados a dados.

```
const [hasData, setHasData] = useState(true);
```

Minhas principais dificuldades na remoção do code smell são: decidir se este estado

deveria ser agrupado com estados de UI ou manter isolado por controlar dados, não apresentação.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: criei objeto tipado preparado para expansão futura, usando Type Inference do TypeScript.

```
type TableDataState = {
  hasData: boolean;
};

const [dataState, setDataState] = useState<TableDataState>({
  hasData: true,
});

const handleDataChange = (newHasData: boolean) => {
  setDataState(prev => ({ ...prev, hasData: newHasData }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 94-95 do `component-token.tsx`. Encontrei 2 estados booleanos (`yScroll`, `ellipsis`) controlando opções de exibição da tabela.

```
const [ellipsis, setEllipsis] = useState(false);
const [yScroll, setYScroll] = useState(false);
```

Minhas principais dificuldades na remoção do code smell são: atualizar handlers que modificam esses valores e garantir compatibilidade com os Switches do Form.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: consolidação em objeto tipado usando Type Inference, reduzindo useState e facilitando manutenção.

```tsx
type TableDisplayOptions = {
  yScroll: boolean;
  ellipsis: boolean;
};

const [displayOptions, setDisplayOptions] = useState<TableDisplayOptions>({
  yScroll: false,
  ellipsis: false,
});

const handleYScrollChange = (enable: boolean) => {
  setDisplayOptions(prev => ({ ...prev, yScroll: enable }));
};

const handleEllipsisChange = (enable: boolean) => {
  setDisplayOptions(prev => ({ ...prev, ellipsis: enable }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 8-9 do `external-panel.tsx` (DatePickerDemo). Identifiquei 2 estados booleanos controlando visibilidade de dropdown e painel.

```tsx
const [visible, setVisible] = React.useState(false);
const [panelVisible, setPanelVisible] = React.useState(false);
```

Minhas principais dificuldades na remoção do code smell são: sincronizar lógica onde `panelVisible` deve resetar quando `visible` é false, usando setState com callback.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: consolidação em objeto tipado usando Type Inference e callback do setState para acessar estado anterior.

```tsx
type DropdownState = {
  visible: boolean;
  panelVisible: boolean;
};

const [dropdownState, setDropdownState] = React.useState<DropdownState>({
  visible: false,
  panelVisible: false,
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 92-93 do `external-panel.tsx` (RangePickerDemo). Encontrei o mesmo padrão do DatePickerDemo com 2 estados booleanos controlando visibilidade.

```tsx
const [visible, setVisible] = React.useState(false);
const [panelVisible, setPanelVisible] = React.useState(false);
```

Minhas principais dificuldades na remoção do code smell são: aplicar a mesma solução do DatePickerDemo mantendo consistência entre os dois componentes.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: reutilização do type `DropdownState` com Type Inference, mantendo padrão consistente entre componentes similares.

```tsx
type DropdownState = {
  visible: boolean;
  panelVisible: boolean;
};

const [dropdownState, setDropdownState] = React.useState<DropdownState>({
  visible: false,
  panelVisible: false,
});
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 80-81 do `dynamic-settings.tsx`. Identifiquei 2 estados booleanos (`bordered`, `loading`) controlando estilos visuais da tabela.

```
const [bordered, setBordered] = useState(false);
const [loading, setLoading] = useState(false);
```

Minhas principais dificuldades na remoção do code smell são: atualizar handlers mantendo compatibilidade com os componentes Switch do formulário.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: consolidação em objeto tipado usando Type Inference do TypeScript, reduzindo useState de 2 para 1.

```
type TableStyleState = {
  bordered: boolean;
  loading: boolean;
};

const [styleState, setStyleState] = useState<TableStyleState>({
  bordered: false,
  loading: false,
});

const handleBorderChange = (enable: boolean) => {
  setStyleState(prev => ({ ...prev, bordered: enable }));
};

const handleLoadingChange = (enable: boolean) => {
  setStyleState(prev => ({ ...prev, loading: enable }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State nas linhas 84-86 do `dynamic-settings.tsx`. Encontrei 3 estados booleanos controlando visibilidade de seções da tabela (título, cabeçalho, rodapé).

```
const [showTitle, setShowTitle] = useState(false);
const [showHeader, setShowHeader] = useState(true);
const [showFooter, setShowFooter] = useState(true);
```

Minhas principais dificuldades na remoção do code smell são: preservar valores iniciais diferentes (false para title, true para header/footer) e atualizar referências no tableProps.

Eu estou usando os seguintes métodos de refatoração para remover o code smell: agrupamento em objeto tipado usando Type Inference, facilitando gerenciamento de visibilidade das seções.

```
type TableSectionVisibility = {
  showTitle: boolean;
  showHeader: boolean;
  showFooter: boolean;
};

const [sectionVisibility, setSectionVisibility] = useState<TableSectionVisibility>({
  showTitle: false,
  showHeader: true,
  showFooter: true,
});

const handleTitleChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showTitle: enable }));
};

const handleHeaderChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showHeader: enable }));
};

const handleFooterChange = (enable: boolean) => {
  setSectionVisibility(prev => ({ ...prev, showFooter: enable }));
};
```

Eu estou atualmente trabalhando na refatoração do seguinte code smell: Multiple Booleans For State na linha 88 do `dynamic-settings.tsx`. Identifiquei o estado `hasData` isolado, preparando para possível expansão de estados relacionados a dados.

```
const [hasData, setHasData] = useState(true);
```

Minhas principais dificuldades na remoção do code smell são: decidir se agrupar com estados de UI ou manter separado por controlar dados da tabela.

Eu estou usando os seguintes métodos de refatoração para remover o code smell:

criação de objeto tipado usando Type Inference, permitindo expansão futura e mantendo consistência.

```
type TableDataState = {
  hasData: boolean;
};

const [dataState, setDataState] = useState<TableDataState>({
  hasData: true,
});

const handleDataChange = (newHasData: boolean) => {
  setDataState(prev => ({ ...prev, hasData: newHasData }));
};
```