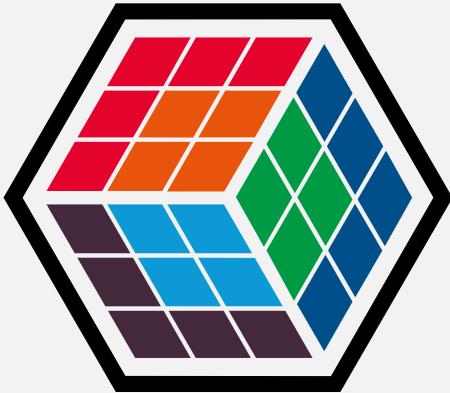


Metaprogramação em JS



TDC POA
2017



STÉFANO
ZANATA
software & design



Programador e Designer?

Stéfano Zanata

10+ anos de experiência

8 com **JavaScript** e 7 com **Node**
Front-end, Ruby, Java, C#

Startups





szanata.com

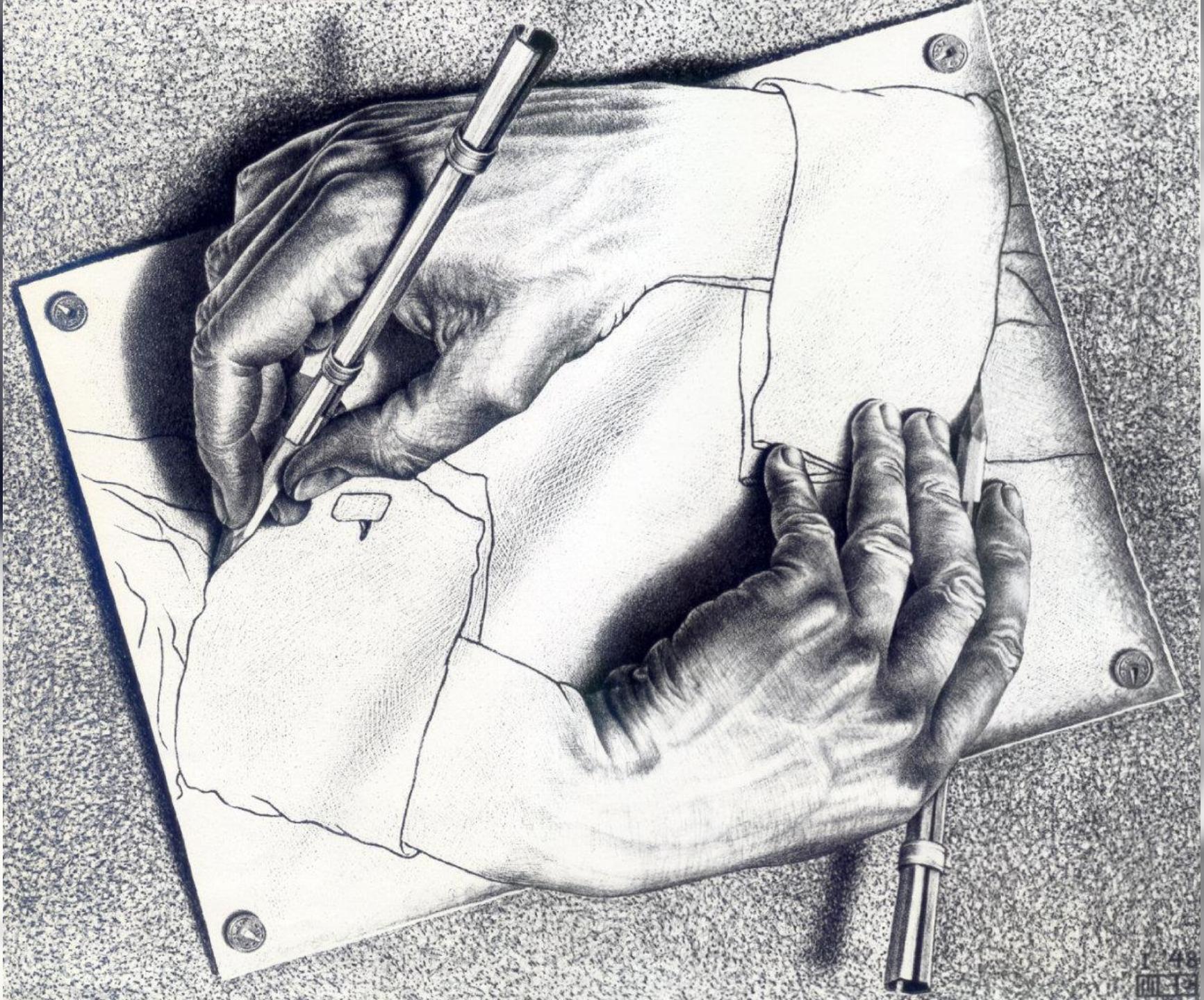


/madeinstefano

Metaprogramação?

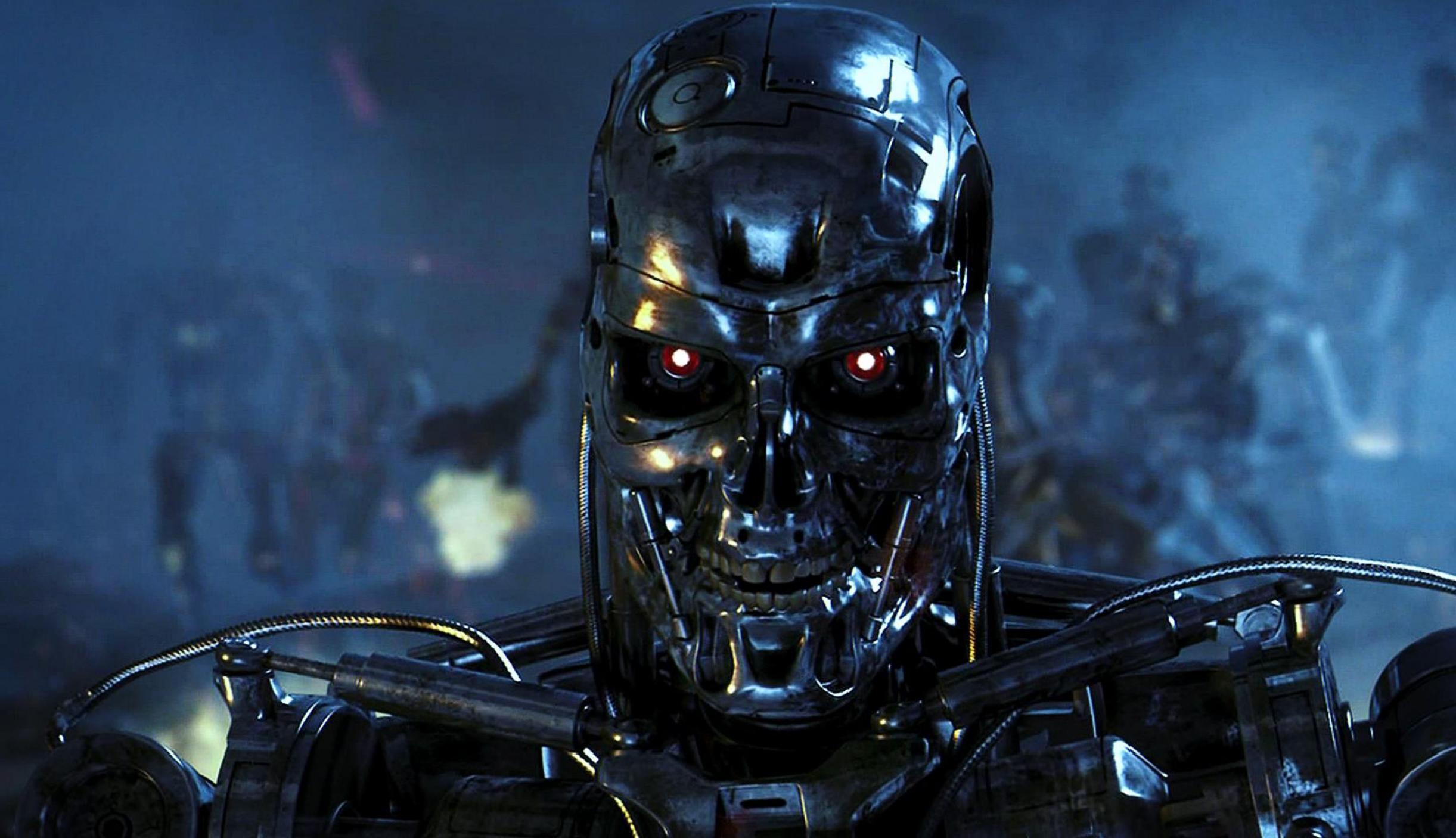


Código que escreve código.



I-48

MM-124



Vamos ver na prática

Exemplo I

Null propagation

```
1  if ( foo.bar ) {  
2      // some actions  
3 }
```

Qual o resultado esperado?



Pode ser **truthy**



Pode ser **falsy**



Pode ser **TypeError: Cannot read property 'bar' of undefined**



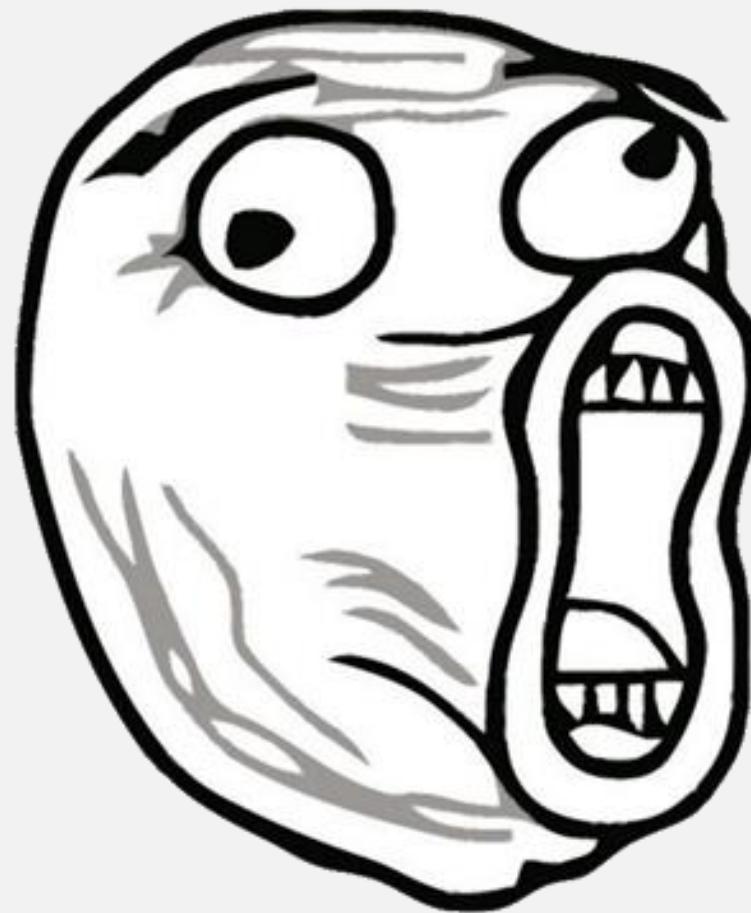
```
1  if ( foo && foo.bar ) {  
2      // some actions  
3 }
```

Problema

Variáveis **undefined** e **null** não permitem **getters**.

É necessário sempre testar quando não se tem controle da origem, como por exemplo **HttpRequest**.

Ok, mas se tivermos múltiplos níveis?



Desvantagens

Cada short-circuit cria duas **branches**

Difícil de **manter**

Difícil de **ler**



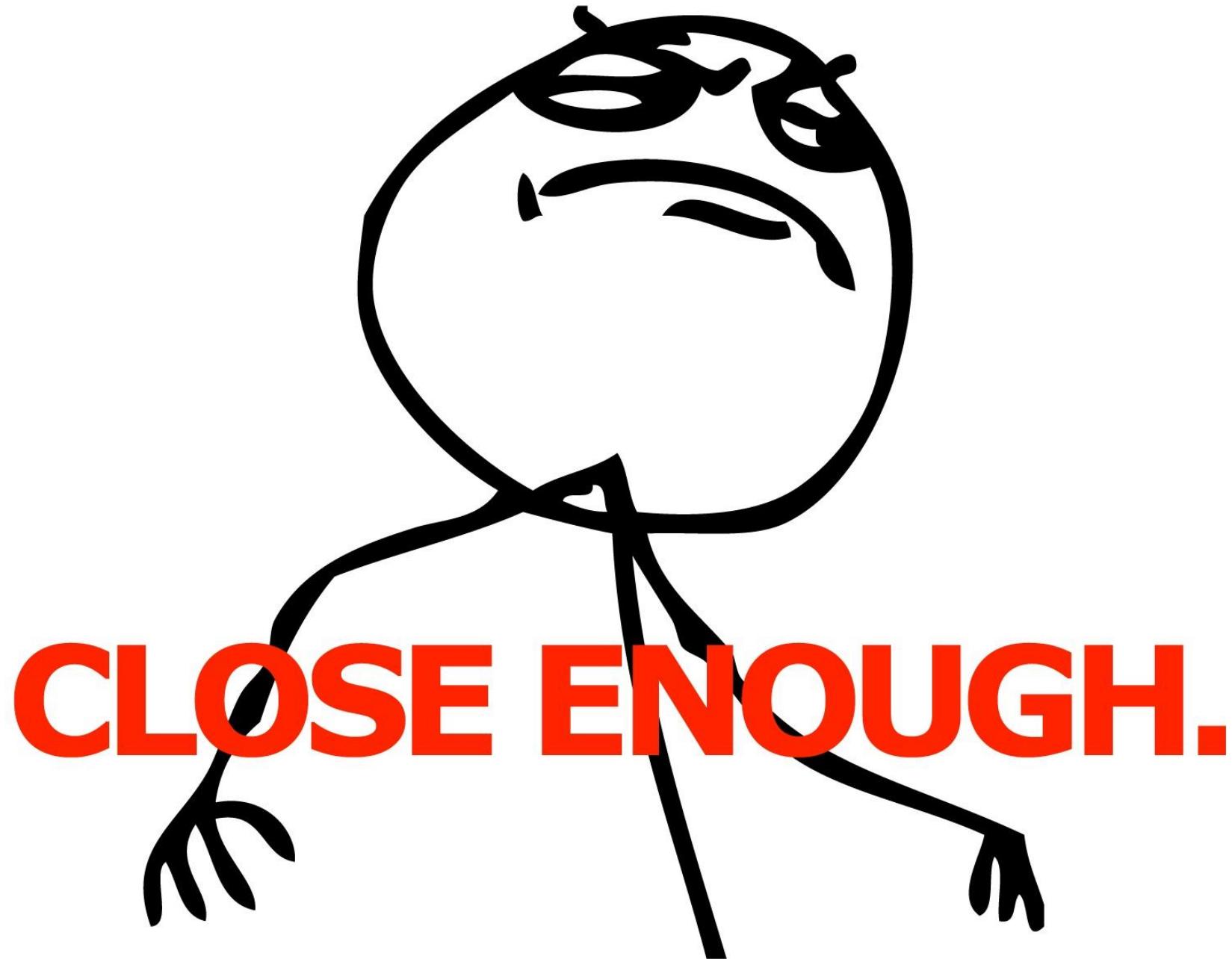
No mundo feliz seria assim

```
1  if ( config.user.geo.coordinates.latitude ) {  
2      // some actions  
3  }
```

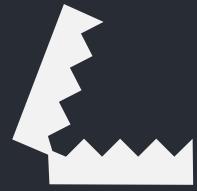


Mas pode ser quase isso

```
1 const config = new SafeObject( rawConfig );
2
3 if ( config.user.geo.coordinates.latitude() ) {
4     // some actions
5 }
```



Qual a mágica?



Traps



IT'S A TRAP

```
1 function proxy( target, property ) {
2     const navigable = target();
3     const fn = () => navigable === undefined ? undefined : navigable[property];
4     return new Proxy( fn, { get: proxy } );
5 }
6
7 module.exports = class SafeObject {
8     constructor( value ) {
9         return new Proxy( () => value, { get: proxy } );
10    }
11};
```

Não sério, funciona...

```
1 const object = new SafeObject( { config: { color: 'red' } } );
2 assert( object.config.color() === 'red' );
3 assert( object.config().color === 'red' );
4
5 const empty = new SafeObject( {} );
6 assert( empty.settings.color() === undefined );
7 assert( empty.settings.geo.coordinates() === undefined );
8
9 const primitive = new SafeObject( 1 );
10 assert( primitive() === 1 );
11
12 const undef = new SafeObject( undefined );
13 assert( undef() === undefined );
```

Vantagens

Clareza de **código**

Apenas uma **branch***

Mais fácil **manter**



Trade-off: se perde um pouco de performance

Conceitos de meta programação utilizados

```
1 function proxy( target, property ) {
2     const navigable = target();
3     const fn = () => navigable === undefined ? undefined : navigable[property];
4     return new Proxy( fn, { get: proxy } );
5 }
6
7 module.exports = class SafeObject {
8     constructor( value ) {
9         return new Proxy( () => value, { get: proxy } );
10    }
11};
```

```
1 function proxy( target, property ) {  
2     const navigable = target();  
3     const fn = () => navigable === undefined ? undefined : navigable[property];  
4     return new Proxy( fn, { get: proxy } );  
5 }  
6  
7 module.exports = class SafeObject {  
8     constructor( value ) {  
9         return new Proxy( () => value, { get: proxy } );  
10    }  
11};
```

Intersecção

Intersecção

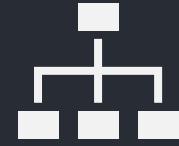
Exemplo II

Generic conditions evaluator

Problema

Dado nome de **rede social** e uma **feature**, um usuário pode saber se ela existe naquela rede social.

```
1 function canUseFeature( network, feature ) {
2   if ( network === 'twitter' ) {
3     return [ 'postImage', 'updateStatus' ].includes( feature );
4   }
5
6   if ( network === 'instagram' ) {
7     return [ 'postImage' ].includes( feature );
8   }
9
10  if ( network === 'facebook' ) {
11    return [ 'postImage', 'updateStatus', 'friendRequest' ].includes( feature );
12  }
13
14  if ( network === 'linkedin' ) {
15    return [ 'updateProfile', 'updateStatus' ].includes( feature );
16  }
17
18  if ( network === 'snapchat' ) {
19    return [ 'postImage', 'postVideo' ].includes( feature );
20  }
21
22  if ( network === 'googleplus' ) {
23    return [ 'postImage', 'updateStatus', 'updateProfile' ].includes( feature );
24  }
25
26  return false;
27 }
```



Muitas branches

Problema

Se precisar adicionar mais uma rede social?

Não isola **domínio** e algoritmo.

Difícil de **testar**.

Difícil de **ler**.

Pequenas mudanças podem impactar outras **branches**.

Alternativa?

```
1 const features = {  
2   twitter: [ 'postImage', 'updateStatus' ],  
3   instagram: [ 'postImage' ],  
4   facebook: [ 'postImage', 'updateStatus', 'friendRequest' ],  
5   linkedin: [ 'updateProfile', 'updateStatus' ],  
6   snapchat: [ 'postImage', 'postVideo' ],  
7   googleplus: [ 'postImage', 'updateStatus', 'updateProfile' ]  
8 };  
9  
10 function canUseFeature( network, feature ) {  
11   return features[network].includes( feature );  
12 }
```

Vantagens

Generalização, apenas uma **branch**.

DRY

Mais fácil de **testar**.

Mais fácil de ler, não lê o código, só a **configuração**.

Isola o **domínio**.



Essa é muito fácil
mesmo, faça
outra mais difícil.

Problema

Dado nome de **rede social**, uma **feature**, e o número de uma **versão**, um usuário pode saber se ela existe naquela rede social.

```
1 function canUseFeature( network, feature, version ) {
2   if ( network === 'twitter' ) {
3     if ( version === 1 ) {
4       return [ 'postImage', 'updateStatus' ].includes( feature );
5     } else if ( version === 2 ) {
6       return [ 'postImage', 'updateStatus', 'postVideo' ]
7         .includes( feature );
8     }
9   }
10
11  if ( network === 'instagram' ) {
12    if ( version === 1 ) {
13      return [ 'postImage' ].includes( feature );
14    } else if ( version === 2 ) {
15      return [ 'postImage', 'postVideo' ].includes( feature );
16    } else if ( version === 3 ) {
17      return [ 'postImage', 'updateStatus', 'postVideo' ]
18        .includes( feature );
19    }
20  }
21}
```

```
22 if ( network === 'facebook' ) {
23   return [ 'postImage', 'updateStatus', 'friendRequest' ]
24     .includes( feature );
25 }
26
27 if ( network === 'linkedin' ) {
28   return [ 'updateProfile', 'updateStatus' ].includes( feature );
29 }
30
31 if ( network === 'snapchat' ) {
32   if ( version < 3 ) {
33     return [ 'postImage', 'postVideo' ].includes( feature );
34   }
35   return [ 'postImage', 'postVideo', 'useEffect' ].includes( feature );
36 }
37
38 if ( network === 'googleplus' ) {
39   if ( version === 4 ) {
40     return [ 'postImage', 'updateStatus', 'updateProfile' ]
41       .includes( feature );
42   } else if ( version > 3 ) {
43     return [ 'postImage' ].includes( feature );
44   }
45 }
```



Problemas

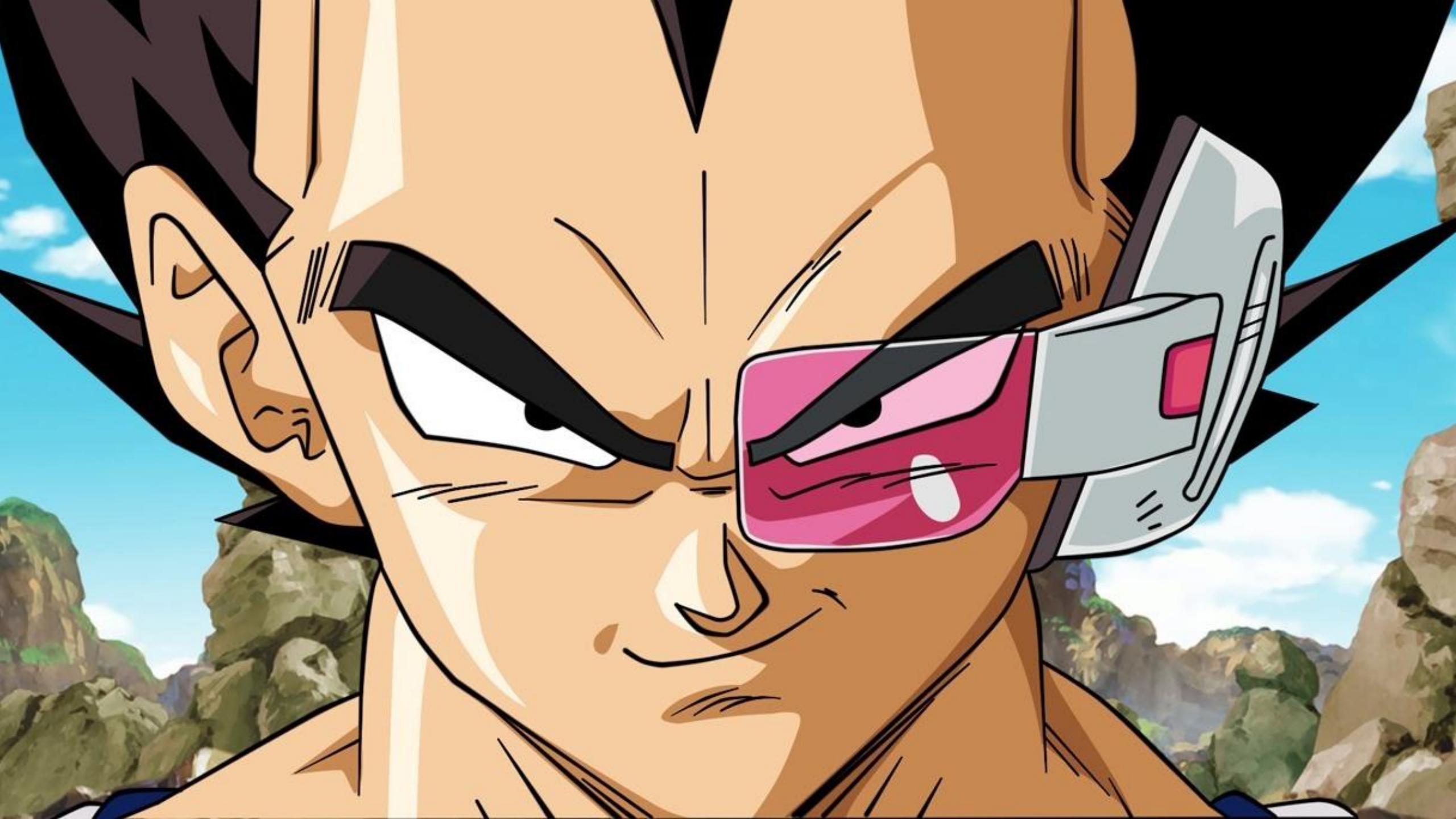
Leitura e **entendimento**.

Muitas **branches**, cobertura de código.

Cenários de **testes**.

Manutenção

Basicamente tudo que havia de **ruim**,
9000 vezes pior.



Alternativa?

Alternativa

E se pudéssemos configurar as features que cada versão das redes sociais tem, **independente** da função que computa elas?

```
1 const features = {  
2   twitter: {  
3     1: [ 'postImage', 'updateStatus' ],  
4     2: [ 'postImage', 'updateStatus', 'postVideo' ]  
5   },  
6   instagram: {  
7     1: [ 'postImage' ],  
8     2: [ 'postImage', 'postVideo' ],  
9     3: [ 'postImage', 'updateStatus', 'postVideo' ]  
10  },  
11  facebook: [ 'postImage', 'updateStatus', 'friendRequest' ],  
12  linkedin: [ 'updateProfile', 'updateStatus' ],  
13  snapchat: {  
14    '<3': [ 'postImage', 'postVideo' ],  
15    '>=3': [ 'postImage', 'postVideo', 'useEffect' ]  
16  },  
17  googleplus: {  
18    '>3': [ 'postImage' ],  
19    4: [ 'postImage', 'updateStatus', 'updateProfile' ]  
20  }  
21}
```



DSL

Vantagens

Ao invés de codificar se **configura**.

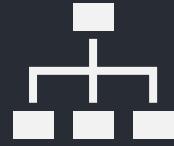
A função tem menos **branchs**.

Fácil de **ler**.

Fácil de **manter**.

Fácil de **testar**.

```
1 function canUseFeature( network, feature, version ) {
2   const rules = features[network];
3
4   if ( Array.isArray( rules ) ) { return rules.includes( feature ); }
5   if ( typeof rules !== 'object' ) { return false; }
6   if ( !isFinite( version ) || !/^\\d$/.test( version ) ) { return false; }
7
8   const safeV = JSON.parse( `{"v": ${version}}` ).v;
9   const expressions = Reflect.ownKeys( rules )
10    .map( k => ( /^\\d+/.test( k ) ? '===' : '' ) + k )
11    .sort( op => !op.startsWith( '=' ) );
12
13   const key = expressions.find( op => Function( `return ${safeV} ${op}` )() );
14
15   return rules[key.replace( /\\d+/g, '' )].includes( feature );
16 }
```



Apenas 2 branchs*

Conceitos de meta programação utilizados

```
1 function canUseFeature( network, feature, version ) {
2   const rules = features[network];
3
4   if ( Array.isArray( rules ) ) { return rules.includes( feature ); }
5   if ( typeof rules !== 'object' ) { return false; }
6   if ( !isFinite( version ) || !/^\\d$/.test( version ) ) { return false; }
7
8   const safeV = JSON.parse( `{"v": ${version}}` ).v;
9   const expressions = Reflect.ownKeys( rules )
10    .map( k => ( /^\\d+/.test( k ) ? '===' : '' ) + k )
11    .sort( op => !op.startsWith( '=' ) );
12
13   const key = expressions.find( op => Function( `return ${safeV} ${op}` )() );
14
15   return rules[key.replace( /\\d+/g, '' )].includes( feature );
16 }
```

```
1 function canUseFeature( network, feature, version ) {
2   const rules = features[network];
3
4   if ( Array.isArray( rules ) ) { return rules.includes( feature ); }
5   if ( typeof rules !== 'object' ) { return false; }
6   if ( !isFinite( version ) || !/\d/.test( version ) ) { return false; } Introspecção
7
8   const safeV = JSON.parse( `{"v": ${version}}` ).v; Geração de código
9   const expressions = Reflect.ownKeys( rules )
10    .map( k => ( /\d/.test( k ) ? '===' : '' ) Introspecção
11    .sort( op => !op.startsWith( '=' ) );
12
13  const key = expressions.find( op => Function( `return ${safeV} ${op}` )() );
14 Geração de código
15  return rules[key.replace( /===/, '' )].includes( feature );
16 }
```

Exemplo III

Dynamic value assignment

Problema

É necessário **atualizar** alguns componentes na tela sempre que o usuário faz uma ação.

Estes valores são totalizados e **somados aos já existentes**.

Eles vêm através de uma **HttpRequest**.

```
1 const payload = {  
2   keywords: [  
3     { value: 7 },  
4     { value: 2 },  
5     { value: 1 }  
6   ],  
7   urls: [  
8     { value: 4 },  
9     { value: 3 },  
10    { value: 2 },  
11    { value: 1 },  
12    { value: 0 }  
13  ],  
14  media: [  
15    { value: 11 },  
16    { value: 7 }  
17  ]  
18};
```

Expectativa

```
1 const knownValues = {  
2     keywords: 23,  
3     urls: 11,  
4     media: 12  
5 };  
6  
7 const result = updateValues( knownValues, payload );  
8  
9 expect( result.keywords ).to.eql( 33 );  
10 expect( result.urls ).to.eql( 21 );  
11 expect( result.media ).to.eql( 30 );
```

Detalhes

O payload pode estar **incompleto**.

Valores na tela podem estar **null** ou
undefined.

```
1 function updateValues( values, newValues ) {
2   const updated = Object.assign( {}, values );
3   if ( newValues.keywords ) {
4     updated.keywords = newValues.keywords
5       .reduce( ( sum, item ) => sum + item.value, updated.keywords || 0 );
6   }
7
8   if ( newValues.urls ) {
9     updated.urls = newValues.urls
10    .reduce( ( sum, item ) => sum + item.value, updated.urls || 0 );
11  }
12
13  if ( newValues.media ) {
14    updated.media = newValues.media
15      .reduce( ( sum, item ) => sum + item.value, updated.media || 0 );
16  }
17
18  return updated;
19 }
```

Problemas

Legitibilidade

Muitas **branches**

Difícil **manutenção**



Mas o principal: DRY

Alternativa?

```
1 function updateValues( values, newValues ) {
2   const updated = Reflect.ownKeys( newValues ).reduce( ( object, key ) => {
3     const value = newValues[key]
4       .reduce( ( sum, item ) => sum + item.value, values[key] || 0 );
5     return Object.assign( object, { [key]: value } );
6   }, { } );
7   return Object.assign( {}, values, updated );
8 }
```

Vantagens

Sem **repetição**.

Menos **branchs**.

Simples para **testar**.

Mas coisas nem sempre são simples assim.

```
1 const payload = {  
2   top_keys: [  
3     { value: 7 },  
4     { value: 2 },  
5     { value: 1 }  
6   ],  
7   deep_links: [  
8     { value: 4 },  
9     { value: 3 },  
10    { value: 2 },  
11    { value: 1 },  
12    { value: 0 }  
13  ],  
14  photos: [  
15    { value: 11 },  
16    { value: 7 }  
17  ]  
18};
```

Payload e modelo não tem as mesmas chaves.

```
1 function updateValues( values, newValues ) {
2   const updated = Object.assign( {}, values );
3   if ( newValues.top_keys ) {
4     updated.keywords = newValues.top_keys
5       .reduce( ( sum, item ) => sum + item.value, updated.keywords || 0 );
6   }
7
8   if ( newValues.deep_links ) {
9     updated.urls = newValues.deep_links
10    .reduce( ( sum, item ) => sum + item.value, updated.urls || 0 );
11  }
12
13  if ( newValues.photos ) {
14    updated.media = newValues.photos
15      .reduce( ( sum, item ) => sum + item.value, updated.media || 0 );
16  }
17
18  return updated;
19 }
```

Um novo problema

Além de todas as desvantagens de antes, temos ainda um **dicionário** envolvido no meio código.

Uma nova abordagem

```
1 const map = new Map( [
2   [ 'keywords', 'top_keys' ],
3   [ 'urls', 'deep_links' ],
4   [ 'media', 'photos' ]
5 ] );
6
7 function updateValues( values, newValues ) {
8   const updated = {};
9   for ( const [ kLeft, kRight ] of map ) {
10     if ( !Reflect.has( newValues, kRight ) ) { continue; }
11     updated[kLeft] = newValues[kRight]
12       .reduce( ( sum, item ) => sum + item.value, values[kLeft] );
13   }
14   return Object.assign( {}, values, updated );
15 }
```



Vantagens

O dicionário fica **isolado**.

Código mais genérico com **duas branches**.

Possível de testar o dicionário e motor **separadamente**.

Conceitos de meta programação utilizados

```
1 const map = new Map( [
2   [ 'keywords', 'top_keys' ],
3   [ 'urls', 'deep_links' ],
4   [ 'media', 'photos' ]
5 ] );
6
7 function updateValues( values, newValues ) {
8   const updated = {};
9   for ( const [ kLeft, kRight ] of map ) {
10     if ( !Reflect.has( newValues, kRight ) ) { continue; }
11     updated[kLeft] = newValues[kRight]
12       .reduce( ( sum, item ) => sum + item.value, values[kLeft] );
13   }
14   return Object.assign( {}, values, updated );
15 }
```

```
1 const map = new Map( [  
2   [ 'keywords', 'top_keys' ],  
3   [ 'urls', 'deep_links' ],  
4   [ 'media', 'photos' ]  
5 ] );  
6  
7 function updateValues( values, newValues ) {  
8   const updated = {};  
9   for ( const [ kLeft, kRight ] of map ) {  
10     if ( !Reflect.has( newValues, kRight ) ) { continue; }  
11     updated[kLeft] = newValues[kRight]    Introspecção  
12       .reduce( ( sum, item ) => sum + item.value, values[kLeft] );  
13   }  
14   return Object.assign( {}, values, updated );  
15 }
```

Exemplo IV

Questão de Prova

Questão 23 (1 pt)

A fim de criar um site de poker online, um programador deve criar um algoritmo para representar um baralho cartas. Este deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

```
1 const ALL_CARDS = [
2   '🂠', '🂡', '🂢', '🂣', '🂤', '🂥', '🂦', '🂧', '🂨', '🂩', '🂪', '🂫', '🂬', '🂭', '🂮',
3   '🂱', '🂲', '🂳', '🂴', '🂵', '🂶', '🂷', '🂸', '🂹', '🂺', '🂻', '🂼', '🂽', '🂾', '🂿',
4   '🂱', '🂲', '🂳', '🂴', '🂵', '🂶', '🂷', '🂸', '🂹', '🂺', '🂻', '🂼', '🂽', '🂾', '🂿',
5   '🂱', '🂲', '🂳', '🂴', '🂵', '🂶', '🂷', '🂸', '🂹', '🂺', '🂻', '🂼', '🂽', '🂾', '🂿'
6 ];
```

Expectativa

```
1 const deck = new Deck();
2
3 expect( deck.length ).to.eql( 52 );
4 expect( deck.draw() ).to.be.a.string;
5 expect( deck.length ).to.eql( 51 );
6
7 const beforeShuffleOrder = Array.from( deck ).join( '' );
8 deck.shuffle();
9 const afterShuffleOrder = Array.from( deck ).join( '' );
10
11 expect( deck.length ).to.eql( 52 );
12 expect( beforeShuffleOrder ).to.not.eql( afterShuffleOrder );
13
14 let forOfLoops = 0;
15 for ( const card of deck ) { forOfLoops++; }
16 expect( forOfLoops ).to.eql( 52 );
17
18 let forEachLoops = 0;
19 deck.forEach( card => forEachLoops++ );
20 expect( forEachLoops ).to.eql( 52 );
```

Assim...



Herdar de Array

Porém...

Questão 23 (1 pt)

A fim de criar um site de poker online, um programador deve criar um classe para representar um baralho cartas. Esta classe deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

Questão 23 (1 pt)

A fim de criar um site de poker online, um programador deve criar um classe para representar um baralho cartas. Esta classe deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

```
1 const deck = new Deck();
2
3 console.log( deck.push( '🂡' ) ); // 53
4
5 deck.splice( 0, 52 );
6
7 console.log( deck.length ); // 1
```

Então...



Array Wrapper

```
1 const cardsSym = Symbol( 'cards' );
2 class Deck {
3     constructor() {
4         this.shuffle();
5     }
6     forEach( fn ) {
7         return Reflect.apply( [].forEach, this[cardsSym], [ fn ] );
8     }
9     get length() {
10        return this[cardsSym].length;
11    }
12    shuffle() {
13        this[cardsSym] = ALL_CARDS.slice().sort( () => 0.5 - Math.random() );
14    }
15    draw() {
16        return this[cardsSym].pop();
17    }
18    *[Symbol.iterator]() {
19        yield* this[cardsSym];
20    }
21 }
```


Contudo...

Questão 23 (1 pt)

A fim de criar um site de poker online, um programador deve criar um classe para representar um baralho cartas. Esta classe deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

Questão 23 (1 pt)

A fim de criar um site de poker online, um programador deve criar um classe para representar um baralho cartas. Esta classe deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

```
1 const deck = new Deck();
2 const cards = Object.getOwnPropertySymbols( deck )[0];
3
4 deck[cards].push( '🂱' );
5
6 assert( deck[cards].length === 53 );
7 assert( deck[cards] instanceof Array );
```

Finalmente...



Private Array Wrapper

```
1 class Deck {  
2     constructor() {  
3         let cards = [];  
4  
5         this.shuffle = () => {  
6             cards = ALL_CARDS.slice().sort( () => 0.5 - Math.random() );  
7         };  
8         this.draw = () => cards.pop();  
9         this.forEach = fn => Reflect.apply( [].forEach, cards, [ fn ] );  
10        this[Symbol.iterator] = function* () {  
11            yield* cards;  
12        };  
13  
14        Object.defineProperty( this, 'length', {  
15            get() { return cards.length; }  
16        } );  
17  
18        this.shuffle();  
19    }  
20}
```

```
1 const deck = new Deck();
2
3 console.log( Reflect.ownKeys( deck ) );
4 // [ 'shuffle', 'draw', 'Length', 'forEach', Symbol(Symbol.iterator) ]
```



Outro jeito?

```
1 class Deck {  
2   constructor() {  
3     let cards = [];  
4  
5     Object.defineProperties( this, {  
6       shuffle: {  
7         enumerable: true,  
8         writable: false,  
9         configurable: false,  
10        value() {  
11          cards = ALL_CARDS.slice().sort( () => 0.5 - Math.random() );  
12        }  
13      },  
14      draw: {  
15        enumerable: true,  
16        writable: false,  
17        configurable: false,  
18        value: () => cards.pop()  
19      },  
20    });  
21  }  
22  
23  const ALL_CARDS = [ ... ]  
24  
25  const card = new Deck();  
26  
27  console.log(card.shuffle());  
28  console.log(card.draw());  
29
```

```
20     forEach: {
21         enumerable: true,
22         writable: false,
23         configurable: false,
24         value: fn => Reflect.apply( [].forEach, cards, [ fn ] )
25     },
26     length: {
27         enumerable: true,
28         configurable: false,
29         get: () => cards.length
30     },
31     [Symbol.iterator]: {
32         enumerable: true,
33         writable: false,
34         configurable: false,
35         *value() { yield* cards; }
36     }
37 } );
38
39     this.shuffle();
40 }
41 }
```

Muito bom!

1

Questão 23 (1 pt)

Afim de criar um site de poker online, um programador deve criar um classe para representar um baralho cartas. Esta classe deve:

1. Conter um conjunto com uma unidade de cada carta do baralho (52).
2. As cartas devem estar organizadas aleatoriamente.
3. Ter um método “compra” para sacar a carta de cima.
4. Ter um método “embaralha” para recolocar todas as cartas no baralho e mudar sua ordem.
5. Permitir ser iterado como uma coleção.
6. Não deve expor métodos desnecessários.
7. Não deve permitir que a coleção de cartas seja alterada.

Conceitos de meta programação utilizados

```
1 class Deck {  
2     constructor() {  
3         let cards = [];  
4  
5         Object.defineProperties( this, {  
6             shuffle: {  
7                 enumerable: true,  
8                 writable: false,  
9                 configurable: false,  
10                value() {  
11                    cards = ALL_CARDS.slice().sort( () => 0.5 - Math.random() );  
12                }  
13            },  
14            draw: {  
15                enumerable: true,  
16                writable: false,  
17                configurable: false,  
18                value: () => cards.pop()  
19            },
```

```
1 class Deck {  
2     constructor() {  
3         let cards = [];  
4  
5         Object.defineProperties(this, {  
6             shuffle: {  
7                 enumerable: true,  
8                 writable: false,  
9                 configurable: false,  
10                value() {  
11                    cards = ALL_CARDS.slice().sort( () => 0.5 - Math.random() );  
12                }  
13            },  
14            draw: {  
15                enumerable: true,  
16                writable: false,  
17                configurable: false,  
18                value: () => cards.pop()  
19            },  
20        });  
21    }  
22  
23    shuffle() {  
24        return this.cards;  
25    }  
26  
27    draw() {  
28        return this.cards.pop();  
29    }  
30  
31    static get ALL_CARDS() {  
32        return [...];  
33    }  
34}
```

```
20     forEach: {
21         enumerable: true,
22         writable: false,
23         configurable: false,
24         value: fn => Reflect.apply( [].forEach, cards, [ fn ] )
25     },
26     length: {
27         enumerable: true,
28         configurable: false,
29         get: () => cards.length
30     },
31     [Symbol.iterator]: {
32         enumerable: true,
33         writable: false,
34         configurable: false,
35         *value() { yield* cards; }
36     }
37 } );
38
39     this.shuffle();
40 }
41 }
```

```
20     forEach: {
21         enumerable: true,
22         writable: false,
23         configurable: false,
24         value: fn => Reflect.apply( [].forEach, cards, [ fn ] )
25     },
26     length: {
27         enumerable: true,
28         configurable: false,
29         get: () => cards.length
30     },
31     [Symbol.iterator]: {
32         enumerable: true, Auto modificação
33         writable: false,
34         configurable: false,
35         *value() { yield* cards; }
36     }
37 } );
38
39     this.shuffle();
40 }
41 }
```

Introspecção

Auto modificação

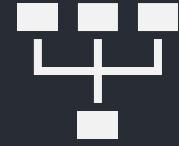
Teoria

Metaprogramação

Por que usar este recurso?

A photograph of Silvio Santos, a Brazilian media tycoon, smiling and speaking. He is wearing a dark blue pinstripe suit, a white shirt, and a blue patterned tie. A small microphone is attached to his lapel. A white speech bubble is positioned to the right of his head, containing the text "Posso perguntar?".

Posso perguntar?



Generalização

Generalização

[...] generalização pode ser entendida como a transformação de um componente específico do domínio em um componente genérico (meta-programa) que é mais amplamente utilizável e reutilizável que o original.

Štuikys & Damaševičius (2013)

Tipos

Estática

Geradores de código

Java

C++

Dinâmica

Código que se auto manipula em tempo de execução

Ruby

JS

Conceitos

Conceitos

Reflexão

Introspecção

Auto modificação (implementação)

Intersecção

Metadados

Geração de código

Ferramentas



Symbols

Symbols

Não são strings!

Metaprogramação por **implementação**

São um novo tipo **primitivo**

Podem ser **únicos**

Podem ser **compartilhados**

Symbols

Existem também os **Well known symbols**, os quais podem ser usados para sobrescrever o que costumava ser interno da linguagem.

Symbols

Alguns principais s̄o: iterator,
hasInstance, toPrimitive...



Reflect

Reflect

Conjunto de ferramentas para fazer
introspecção em objetos.

Para cada possibilidade de **Proxy**,
existe um método de Reflect.

Reflect

Muitas das funções são as mesmas do **Object**.

Foi criado para isolar métodos que não precisam de um objeto para serem usados.

Reflect

Alguns principais são: get, set, apply, ownKeys, construct...



Proxy

Proxy

Serve para criação de **traps**, basicamente interceptar e modificar o comportamento de operações fundamentais.

Proxy

Principais operações interceptáveis:
get, apply, construct, set...



Object

Object

Possuí um conjunto de ferramentas tanto pra **introspecção**, quanto para **implementação**.

Object

Ainda permite clonar objetos, criar object, etc.

Não é recente, diferente das outras ferramentas.

Object

Principais funções: defineProperty,
create, assign...

Obrigado!





/madeinstefano