xib2swift: do legado *Interface Builder* para Swift

Vinicius Caputo, Ricardo Terra
Departamento de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
[vinicius.@estudante,terra]@ufla.br

ABSTRACT

Apple has traditionally advocated for the utilization of Interface Builder, a tool designed to structure user interface components within XIB files. However, it has become apparent that Interface Builder is associated with significant drawbacks, such as conflicts in code versioning systems and prolonged project compilation times, which have rendered the approach increasingly unsustainable. Given the ongoing shift by numerous IT companies towards Swift, this paper introduces `xib2swift`—a tool specifically designed to facilitate the transition from Interface Builder to Swift for iOS development projects. This research evaluates `xib2swift` through a dual approach: firstly, implementing the tool in a major financial corporation's project resulted in a nearly 83% reduction in build time. Secondly, feedback gathered via a questionnaire from developers across different teams corroborates the view that while manual conversion is feasible, it tends to be tedious and susceptible to errors. Significantly, all developers concurred that the tool proved beneficial and effective for conversion purposes and expressed intentions to utilize it in future projects.

Demo: https://doi.org/10.6084/m9.figshare.25948927

1 INTRODUÇÃO

O desenvolvimento de aplicações para iOS evoluiu significativamente desde o lançamento do sistema operacional pela Apple. Inicialmente, a Apple adaptou seu kit de desenvolvimento de software MacOS, conhecido como Cocoa, para suportar interfaces táteis sob o novo nome de Cocoa Touch [10]. O ambiente de desenvolvimento integrado (IDE) da Apple, o Xcode, incorpora o Interface Builder, uma ferramenta projetada para facilitar a criação visual de interfaces de usuário para aplicativos [4].

No *Interface Builder*, é possível arrastar componentes pré-definidos, como botões e caixas de texto, e personalizá-los conforme necessário, permitindo a construção de elementos personalizados [5]. Toda essa estrutura visual da tela, juntamente com os componentes, é armazenada em arquivos XIB [2, 3, 9].

No entanto, apesar de sua utilidade, o Interface Builder e sua evolução, o Storyboard, que permite a definição de múltiplas telas em um único arquivo [6], começaram a mostrar limitações significativas ao longo do tempo. Problemas como conflitos em sistemas de controle de versão e aumento significativo no tempo de compilação dos projetos tornaram essas ferramentas um desafio para a sustentabilidade do desenvolvimento ágil [1]. Esses problemas se tornaram particularmente proeminentes à medida que os projetos de aplicativos aumentaram em complexidade e escala.

Diante deste cenário, o mercado tem gradualmente se deslocado para abordagens alternativas que fujam desses problemas [8]. Uma das principais alternativas é a migração para o desenvolvimento exclusivo em Swift, abandonando o uso do Interface Builder. Este artigo apresenta `xib2swift`, uma ferramenta que simplifica a transição de projetos que utilizam o Interface Builder para uma abordagem puramente em Swift, que é oficialmente a abordagem recomendada pela Apple [7].

Como contribuições relevantes: (i) a avaliação em um projeto de uma grande empresa financeira demonstrou redução de quase 83% no tempo de compilação; e (ii) um questionário aplicado a oito programadores corrobora a visão de que, embora a conversão manual não seja difícil, ela é trabalhosa e suscetível a erros. Significativamente, todos os desenvolvedores concordaram que a ferramenta provou ser benéfica e eficaz para fins de conversão e expressaram intenções de utilizá-la em projetos futuros.

O documento está organizado como a seguir. A Seção 2 descreve detalhadamente a ferramenta `xib2swift`. A Seção 3 discute a avaliação prática desta ferramenta em uma grande empresa financeira, destacando os ganhos de eficiência e a redução do tempo de compilação. Por fim, a Seção 4 conclui e propõe direções para futuras pesquisas e desenvolvimentos na área.

2 FERRAMENTA XIB2SWIFT

A ferramenta xib2swift é uma solução de código aberto[1] desenvolvida e mantida pelo primeiro autor deste artigo para auxiliar projetos no mercado que desejam converter interfaces criadas no *Interface Builder* e armazenadas em arquivos XIB para código Swift.

Para realizar essa conversão, a ferramenta analisa o código da interface presente nos arquivos XML, identificando os pontos necessários para a conversão por meio de análise estática, e os converte em código Swift equivalente passando pelas seguintes etapas:

(1) **Análise do código XIB (*Parsing*)**: Realiza o *parsing* do código XIB, extraindo os "pontos de interesse", que são as partes do XML que são utilizadas nas etapas subsequentes.
(2) **Geração das declarações dos componentes e suas propriedades**: Identifica e converte as informações sobre os componentes, como cor de fundo, fonte e ações de botões. As *tags* que representam essas propriedades são convertidas e manipuladas por funções que realizam a conversão.
(3) **Geração da Hierarquia de Visualização**: Examina a ordem dos componentes no XML e os componentes aninhados dentro de outros para determinar a ordem em que eles são inseridos na tela, mantendo a sobreposição da interface.
(4) **Geração de *Constraints* (Restrições)**: Analisa os fragmentos do XML que afetam o posicionamento dos elementos de *Autolayout*, ou seja, as âncoras, transformando tais fragmentos em código Swift equivalente.

A decisão de desenvolver o projeto como código aberto tem múltiplas motivações, com ênfase no desejo da ampla utilização

[1] https://github.com/vinicius-caputo/xib2swift

da ferramenta e na contribuição deste autor para a comunidade. Além disso, a falta de documentação abrangente sobre a estrutura dos arquivos XIB é um fator significativo, uma vez que, sem um entendimento completo dos possíveis cenários, a ferramenta não pode prever todas as situações. Portanto, o xib2swift foi criado e está em constante aprimoramento. Sempre que ocorre uma conversão incorreta de um componente, é possível relatar o problema na página do projeto, e o autor ou a comunidade trabalharão para adicionar suporte ou encontrar uma solução para esse caso, tornando a ferramenta mais completa e abrangendo o maior número de possíveis cenários.

A arquitetura da ferramenta foi projetada de forma a facilitar a contribuição da comunidade. Em várias etapas da conversão, são utilizados dicionários com o intuito de desacoplar partes que podem ser adicionadas pelos contribuintes, tornando a contribuição mais acessível. Um exemplo prático é um dicionário que contém as propriedades a serem ignoradas. Se um membro da comunidade encontrar alguma propriedade não mapeada que deve ser ignorada, poderá adicioná-la ao objeto e contribuir com o projeto sem a necessidade de compreender todas as etapas internas.

2.1 *Running Example*

Para melhor desenvolver esta seção, foi criado um aplicativo simples utilizando o *Interface Builder*. Partes desse aplicativo são convertidas para seu código equivalente em Swift, as quais são utilizadas para exemplificar e entender melhor cada parte do processo de conversão.

O aplicativo é uma representação simplificada de um jogo da velha conforme ilustrado na Figura 1. O jogo possui apenas uma tela com os seguintes elementos:

- Um rótulo (*label*) que contém o título do jogo.
- Uma *view* de suporte na parte inferior que contém nove botões representando as nove lacunas do tabuleiro, Esses nove elementos possuem *outlets* (i.e., ligações entre o arquivo XIB e o código Swift) com os nomes de *b0* a *b8* para tratar o toque da posição que o jogador escolheu, além de verificar o estado do jogo.
- Um botão abaixo da *view* com os nove botões. Esse elemento também possui *outlet* com o código Swift, chamado de *resetButton* para tratar a ação de reiniciar o jogo.

2.2 Análise do Código XIB (*Parsing*)

Como mencionado anteriormente, o objetivo dessa etapa é extrair as *tags* que são úteis para as etapas subsequentes de conversão. Isso é feito através de análise estática, ou seja, o *parsing* do XIB. Para isso, a ferramenta inicia convertendo o XML em uma estrutura de dados mais facilmente manipulável utilizando um conversor de código aberto amplamente utilizado para *parsing* de HTML (Linguagem de Marcação de HiperTexto) e XML (*Extensible Markup Language*), o *posthtml-parser*[2]. A conversão faz com que cada *tag* possua as seguintes propriedades:

- ***Tag***: Uma representação em forma de *string* do nome da *tag* XML.

[2]https://github.com/posthtml/posthtml-parser

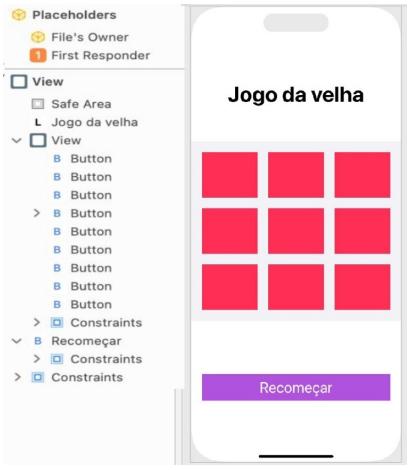

Figura 1: Jogo da velha desenvolvido no *Interface Builder*

```
<constraints>
  <constraint firstAttribute="height" constant="50" id="LFT-MU-Wnj"/>
</constraints>
```

Listagem 1: *Tag* de *constraint* do botão de recomeçar

```
{
    "tag": "constraints",
    "attrs": {},
    "content": [{
        "tag": "constraint",
        "attrs": {
            "firstAttribute": "height",
            "constant": "50",
            "id": "LFT-MU-Wnj"
        }
    }]
}
```

Listagem 2: Estrutura gerada pelo *posthtml-parser*

- ***Attrs***: Um dicionário que armazena os atributos daquela *tag* em formato de chave-valor.
- ***Content***: O conteúdo da *tag*, a qual pode ser representado por outra *tag* com as mesmas propriedades mencionadas ou pode ser uma *string*.

Um exemplo prático é a conversão de uma *tag* de *constraint* presente no XIB do jogo da velha, exemplificado na Listagem 1 para a estrutura que o *posthtml-parser* retorna, como pode ser observado na Listagem 2.

No entanto, a estrutura gerada pelo *posthtml-parser* contém propriedades que não são utilizadas no projeto, como os caracteres de quebra de linha e elementos vazios. Portanto, após essa primeira etapa, a estrutura é convertida para uma estrutura ainda mais simples representada na Listagem 3. Nessa nova estrutura, as propriedades não utilizadas são removidas, e uma referência para o elemento pai (opcional, já que nem todas as *tags* têm um elemento pai) é adicionado para facilitar a navegação nas etapas futuras.

Após a etapa de limpeza, a próxima etapa é obter os "pontos de interesse". Para tal, a ferramenta realiza uma navegação recursiva utilizando a abordagem de busca em largura, a qual é empregada para garantir que os elementos de tela sejam interpretados e armazenados na sequência certa. Isso é necessário para gerar a hierarquia

```typescript
export interface XibNode {
    tag: string,
    attrs: {
        [key: string]: string
    }
    content: XibNode[],
    father?: XibNode
}
```

Listagem 3: Estrutura que representa uma *tag* do XML

de *views* e para processar as *tags* referenciadas como *outlets* antes das demais, uma vez que são armazenadas no topo da árvore. As *tags*, que representam os pontos de interesse, e são armazenadas para utilização nas etapas futuras são:

- *Constraint*: São as *tags* utilizadas para armazenar atributos e propriedades do posicionamento dos elementos como já representado na Listagem 1.
- *Subviews*: São as *tags* referentes às *subviews* de cada componente. Como todo componente herda de *View*, todo XIB possui ao menos uma *tag* desse tipo.
- *Id*: Essa propriedade não se trata de uma *tag*, mas sim, de um atributo presente em todas as *tags* que representam um "ponto de interesse". Esse atributo é empregado para criar o nome exclusivo de cada elemento. Se não o tem, é automaticamente criado um genérico e único.
- *Outlets*: Representam as ligações entre o *Interface Builder* e o código Swift, armazenam o nome que o programador atribuiu a essa propriedade e o identificador da *tag* à qual se referem. Nem todos os componentes são representados com um *outlet*, em sua maioria, são referenciados aqueles que requerem alguma transformação. No entanto, com essa propriedade, é possível, junto com a propriedade que associa um ID a um nome, verificar se algum dos *outlets* faz referência ao ID da *tag* atual. Caso faça, ao invés de atribuir o nome genérico, é usado o nome original fornecido pelo programador para o componente, facilitando ainda mais o processo de conversão. É importante destacar que, como previamente afirmado, todos os *outlets* estejam armazenados antes da análise dos demais elementos.

2.3 Geração das declarações dos componentes e suas propriedades

Nesta etapa, a ênfase está na geração de declarações dos elementos da interface de usuário (UI), ou seja, os componentes e suas propriedades. Para esse propósito, o arranjo que contém todas as *subviews* é utilizado. Conforme mencionado anteriormente, cada *tag subview* pertence a um elemento de tela e cada arquivo XIB tem pelo menos uma *tag* desse tipo, representando a *view* base do XIB. Adicionalmente, incorporam-se outros componentes a essa propriedade para a criação de uma interface mais complexa, cuja composição se reflete na estrutura do XML.

Para gerar as declarações, o processo é dividido em duas etapas: a geração dos atributos da *tag* e a geração dos atributos dos sub-nós.

2.3.1 Geração dos Atributos da tag. Os atributos da *tag* correspondem aos atributos presentes na *tag* do elemento, como pode ser

```
<button opaque="NO" contentMode="scaleToFill"
  contentHorizontalAlignment="center"
  ↪ contentVerticalAlignment="center"
  buttonType="system" lineBreakMode="middleTruncation"
  translatesAutoresizingMaskIntoConstraints="NO" id="Xp5-wO-rxI">
  <rect key="frame" x="19.99" y="20" width="104.33"
    height="85.33"/>
  <color key="backgroundColor" systemColor="systemPinkColor"/>
  <color key="tintColor" white="1" alpha="1" colorSpace="custom"
    customColorSpace="genericGamma22GrayColorSpace"/>
  <state key="normal" title="Button"/>
  <buttonconfiguration key="configuration" style="plain"/>
</button>
```

Listagem 4: *Tag* de um dos botões do jogo da velha

```
lazy var b1: UIButton = {
  let button = UIButton()
  button.contentMode = .scaleToFill
  button.contentHorizontalAlignment = .center
  button.contentVerticalAlignment = .center
  button.translatesAutoresizingMaskIntoConstraints=false
  button.backgroundColor = .systemPink
  button.tintColor = UIColor(cgColor:
  ↪ CGColor(genericGrayGamma2_2Gray: 1, alpha: 1))
  button.configuration = .plain()
  button.setTitle("", for: .normal)
  return button
}()
```

Listagem 5: Conversão de um dos botões para Swift

observado na propriedade `attrs` na Listagem 2. Para cada atributo, existe um par de chave-valor.

O primeiro processo é gerar a chave. No entanto, a forma como a chave é escrita no XML pode ser diferente da representação em Swift. Para superar esse desafio, um dicionário auxiliar é utilizado para mapear as chaves correspondentes entre XML e Swift. Primeiramente, verifica-se se a chave está presente nesse dicionário e, em caso positivo, o nome correspondente é utilizado; caso contrário, a própria chave é empregada. Por exemplo, a propriedade *opaque* (Listagem 4, linha 1), que é um booleano representando se a propriedade é opaca ou não, é chamada *isOpaque* em Swift.

Além disso, a chave passa por outra verificação, caso essa chave precise ser ignorada. É utilizado um dicionário auxiliar de propriedades que devem ser excluídas, uma vez que algumas propriedades não têm relevância na conversão para Swift. Um exemplo disso é a propriedade ID, que está presente em toda *tag* "ponto de interesse", e é usada apenas como referência no arquivo XIB e em outras partes da ferramenta. Ao comparar a Listagem 4 (onde o ID está presente na linha 4) com a Listagem 5, é possível notar que a propriedade não é incluída na conversão.

Após a criação da chave, o próximo passo é gerar o "valor" correspondente ao par chave-valor. A geração do valor envolve o uso de um dicionário para interpretar o par e produzir o resultado correspondente. Essa abordagem se faz necessária devido à impossibilidade de verificar antecipadamente a tipagem do valor no arquivo XIB, que pode ser um número, uma *string*, um booleano, ou enumeração, e cada tipo é representado de forma específica e diferente em Swift.

Primeiramente, é utilizada a seguinte lógica: para a maioria dos valores, assume-se que eles são booleanos, números ou enumerações. A primeira verificação ocorre para identificar se o valor é um

booleano o qual é representado no XIB como *YES* ou *NO*, correspondendo respectivamente a *true* e *false*. Logo depois, é realizado um teste para ver se o valor é composto totalmente por números, se sim o valor é representado como um número sem qualquer modificação. E, por último, caso não seja nenhum dos tipos anteriores, o valor é representado como uma enumeração. As enumerações em Swift são representadas como um ponto seguido do valor como ".textoValor", com o nome da propriedade em *CamelCase*[3].

No entanto, isso não abrange todos os casos, restando apenas um caso: valores de *string*, que são representados entre aspas em Swift. Contudo, é impossível diferenciar *strings* e enumerações pelo XIB, exclusivamente por esse caso a chave é verificada. Se a chave se refere a uma propriedade de *string*, como as chaves *text* ou *placeholder*, o valor é representado entre aspas, como em "texto do valor".

Adicionalmente, após a criação da chave e do valor em Swift, realiza-se uma comparação para verificar se esses valores correspondem aos padrões definidos para a respectiva propriedade em Swift. Um exemplo é o atributo *isOpaque*, que é inicialmente atribuído como *false* por padrão. Caso os valores, após a conversão, coincidam com o valor padrão, eles são omitidos na declaração, uma vez que representam apenas um reforço explícito. Isso fica evidente na transição da Listagem 4 para a Listagem 5 (em que a propriedade não está presente nas linhas 3 a 10), indicando que tal propriedade não foi incluída.

2.3.2 Geração dos Atributos dos sub-nós. A geração dos atributos dos sub-nós ocorre nas *tags* filhas, ou seja, aquelas contidas dentro da *tag* da propriedade que está sendo processada, como as propriedades entre as linhas 5 a 11 da Listagem 4. Todas essas *tags* são processadas, exceto a *tag subview*, caso ela esteja presente. Nesse processo, cada *tag* filha é tratada individualmente por meio de um método que recebe o tipo da *tag* e o atributo do nó que deve ser processado. É o método mais complexo da ferramenta, pois cada nó requer um tratamento específico. Portanto, cada tipo de *tag* possui uma função correspondente que retorna uma *string*, representando a conversão dessa *tag* para Swift.

Para auxiliar nessa conversão, é utilizada uma classe chamada *Resolve*, responsável por converter imagens e cores para a declaração em Swift, já que esses dois elementos podem ser representados de várias maneiras diferentes. Por exemplo, as cores podem ser representadas em RGB, hexadecimal ou ainda como enumerações com cores predefinidas.

Após a geração dessas propriedades, é possível criar a declaração do elemento que está sendo convertido. Para sua realização, utiliza-se a forma de declaração mais utilizada, a notação de variável "*lazy*", uma notação que instancia os elementos apenas quando são necessários, por meio de uma função anônima. O ID da *tag* que foi convertida é associado ao nome que representa esse elemento, seja um nome genérico ou o nome original, e as propriedades convertidas são adicionadas a essa declaração. Como pode ser observado no exemplo de conversão, um dos botões presentes na *view* do jogo, ilustrado na Listagem 4, tem suas propriedades entre as linhas 5 a 11 convertidas para as equivalentes em Swift, representadas nas linhas 7 a 10 da Listagem 5.

[3]CamelCase é uma convenção de nomenclatura em programação que combina palavras juntas sem espaços, capitalizando a primeira letra de cada palavra, exceto a primeira.

```
1 view.addSubview(label__ZAb_Kq_94G)
2 view.addSubview(view__efd_Lt_FBX)
3 view.addSubview(resetButton)
4 view__efd_Lt_FBX.addSubview(b0)
5 view__efd_Lt_FBX.addSubview(b1)
```

Listagem 6: Fragmento da hierarquia de visualização do jogo da velha

2.4 Geração da Hierarquia de Visualização

Para adicionar um elemento à tela em Swift, é necessário incluí-lo no arranjo de *subviews* de cada componente. Como mencionado anteriormente, a ordem em que os componentes são adicionados nesse atributo *subview*, afeta a sobreposição de cada elemento.

Atendendo a esses requisitos, o xib2swift gera a hierarquia de *views* usando os nós armazenados no arranjo da propriedade *subviews* da classe da primeira etapa, a classe que manipula o XIB. Devido ao uso do algoritmo de busca em largura usado para percorrer o XML, é garantido que a ordem dos elementos a serem adicionados esteja correta. Ou seja, o componente pai é adicionado antes do componente filho, com a sobreposição correta.

Logo, para gerar as declarações de hierarquia, é necessário navegar por cada arranjo de *subviews*. Cada declaração é gerada usando como base o componente filho, aquele que está sendo processado, e usando a referência do componente pai acessível por meio do atributo "father" da estrutura da Listagem 3. Dessa forma, as declarações seguem o seguinte padrão:

```
ComponentePai.addSubview(ComponenteFilho)
```

No jogo da velha, é possível perceber de maneira mais clara como a estrutura é gerada. Três componentes são adicionados à *view* base, o *label* que contém o título do jogo, a *view* que contém os botões do jogo, e o botão de recomeçar o jogo. Por meio da Listagem 6, é possível perceber que todos esses componentes são adicionados à *view* base nas linhas 1 a 3, e dois dos botões do jogo presentes na linha 4 e 5 são adicionados a *view* de suporte.

2.5 Geração de *Constraints* (Restrições)

Para gerar as *constraints*, o processo é dividido em duas partes. Primeiramente, é identificado o tipo de *constraint* a ser gerada, com base nas propriedades encontradas. Isso ocorre pois a geração da *constraint* pode variar dependendo das propriedades envolvidas. Após essa identificação, é gerado o código Swift equivalente, agrupando as *constraints* pertencentes ao mesmo elemento. Na segunda parte, as *constraints* são ordenadas para facilitar a visualização do usuário e a legibilidade do código.

As *constraints* em Swift são definidas por meio de funções que fazem referência a outros elementos ou usam um valor numérico fixo. Essa referência é feita no XIB usando os IDs das *tags* dos elementos.

O processo é iniciado verificando se o tipo de *constraint* se refere a um valor numérico fixo. Se for o caso, é gerada a *constraint* atribuindo o valor fixo a esse elemento. Para fazer essa geração, é utilizado o nome do elemento, a âncora e o valor numérico desejado. Por exemplo, a *constraint* do botão de recomeçar apresentado na Listagem 1 que atribui uma altura com proporção fixa de 50 é convertida para o seguinte código em Swift:

```
resetLabel.heightAnchor.constraint(equalToConstant:50)
```

```
<constraints>
  <constraint firstItem="fnl-2z-Ty3" firstAttribute="trailing"
  ↪  secondItem="ZAb-Kq-94G" secondAttribute="trailing"
  ↪  constant="20" id="12K-zx-XM6"/>
  <constraint firstAttribute="bottom" secondItem="efd-Lt-FBX"
  ↪  secondAttribute="bottom" constant="264" id="2Xf-sd-hbI"/>
  <constraint firstItem="efd-Lt-FBX" firstAttribute="leading"
  ↪  secondItem="fnl-2z-Ty3" secondAttribute="leading"
  ↪  id="r5W-hc-z03"/>
</constraints>
```

Listagem 7: Amostra de *constraints* de elementos adicionados a *view* base do jogo da velha em XIB

```
b5.topAnchor.constraint(equalTo: b2.bottomAnchor, constant: 20),
b5.leadingAnchor.constraint(equalTo: b4.trailingAnchor, constant:
↪  20),
b5.widthAnchor.constraint(equalTo: b0.widthAnchor),
b5.heightAnchor.constraint(equalTo: b0.heightAnchor),
```

Listagem 8: *Constraints* de um dos botões em Swift

A segunda forma de geração ocorre quando um elemento faz referência a outro. Nesse caso, é usado o nome e a âncora do primeiro elemento para associar a âncora do segundo elemento. Essas informações são extraídas da *tag* da *constraint*. Existe uma variação em que a propriedade que faz referência ao primeiro elemento pode não estar presente nessa *tag*. Nesse caso, o primeiro elemento faz referência ao componente em que a *tag* está contida, como pode ser observado na linha 3 da Listagem 7.

Também é possível que algumas propriedades adicionais sejam adicionadas nessas funções, como um atributo chamado *constant* e um chamado de *multiplier*. O atributo *constant* é usado para adicionar um valor numérico fixo à *constraint* que está sendo referenciada, como um espaçamento de unidade entre os elementos. Já o atributo *multiplier* é utilizado para aplicar uma proporção ao elemento, por exemplo, um multiplicador de "3/4" define que a propriedade é 75% do valor da constante referenciada.

Após gerar cada *constraint*, elas são armazenadas em um dicionário, onde a chave representa o nome do elemento e o valor é um arranjo das *constraints* para aquele elemento. Isso permite agrupar as *constraints* para cada componente.

Por fim, as *constraints* são ordenadas de acordo com os tipos de âncoras a que se referem. A ordem escolhida é *'top', 'bottom', 'leading', 'trailing', 'centerX', 'centerY', 'width' e 'height'*. Dessa forma, é garantido que as *constraints* sempre tenham a mesma ordem, o que melhora a legibilidade do código gerado e resulta em um código mais limpo, como é ilustrado na Listagem 8 que referencia as *constraints* geradas de um dos botões do jogo.

3 AVALIAÇÃO

Esta seção trata da aplicação e validação da ferramenta criada pelo primeiro autor em um projeto de uma grande empresa brasileira do setor financeiro que optou por anonimato, mas contribuiu com autorização para a coleta dos dados da aplicação da ferramenta em seu sistema.

Desta forma, a Seção 3.1 detalha o sistema em que a ferramenta foi utilizada. A Seção 3.2 consiste em uma análise dos dados obtidos no incremento do desempenho do tempo de *build* do sistema. E, por fim, a Seção 3.3 valida a aplicabilidade da ferramenta sob a perspectiva do desenvolvedor.

3.1 Sistema

O sistema avaliado é um grande módulo de um aplicativo iOS, no qual o padrão de desenvolvimento era a utilização de arquivos XIB para criação das telas. No entanto, ao longo do tempo, esse padrão foi descontinuado devido ao aumento substancial do tempo de *build* e os problemas trazidos pela abordagem que utiliza XIB, como descritos na Seção 1.

Assim, ocorreu uma transição para a abordagem de programar apenas usando Swift. Contudo, o novo método foi aplicado inicialmente apenas aos projetos novos, portanto não eliminou os arquivos XIB remanescentes dos projetos anteriores, o que resultou em uma acumulação de arquivos legados, criando uma necessidade de refatoração.

A ferramenta xib2swift foi criada e utilizada nesse contexto para ajudar a converter arquivos XIB que restaram, permitindo aproveitar as vantagens da nova abordagem. Os dados provenientes dessa utilização são empregados na avaliação da eficácia da ferramenta.

3.2 Economia no Tempo de *Build*

Conforme mencionado na Seção 1, uma das vantagens da transição da abordagem que utiliza o *Interface Builder* para a abordagem exclusivamente utilizando Swift é a redução significativa no tempo de *build* do sistema. Nesta seção, é explorada a avaliação do ganho de desempenho após a conversão do sistema mencionado.

3.2.1 Metodologia. Para realizar essa avaliação, é utilizado o registro de compilação, *build*, do sistema fornecido pelo Xcode, a IDE de desenvolvimento iOS. Esse registro abrange todos os arquivos que foram compilados, incluindo informações sobre o tempo de compilação de cada arquivo.

Para a análise, foi comparado cada arquivo XIB em 24 versões do sistema, todas após a implementação da ferramenta. Comparando cada versão com a subsequente, é identificado quais arquivos foram convertidos naquela versão, a quantidade de linhas dos arquivos XIB originais e dos arquivos Swift após a conversão e, ainda, mensurado o tempo de compilação desses arquivos. Ademais, tomou-se nota da quantidade de arquivos convertidos e dos que ainda precisam de conversão.

3.2.2 Avaliação e Discussão. A análise dos dados da ferramenta ao longo de oito meses no projeto revelou que 165 dos 269 arquivos XIB foram convertidos, representando uma taxa de conversão de 61,33%.

Contudo, após a coleta de dados, observou-se que durante o processo de conversão, os membros do projeto aproveitaram para realizar refatorações nos arquivos, removendo partes que não eram utilizadas, o que afetou os resultados obtidos pela ferramenta, gerando anomalias, como arquivos que ficaram com menos linhas de código após a conversão. Consequentemente, foi necessário conduzir uma nova rodada de coleta de dados, convertendo os arquivos apenas aplicando o código gerado pela ferramenta, sem outras refatorações, evitando resultados alterados.

A partir desses novos dados foi possível analisar que o tempo de compilação de todos os arquivos XIB juntos somavam 93,5 segundos.

Após a conversão para Swift, esses mesmos arquivos compilaram em 16,3 segundos, representando uma melhoria de 82,57%. Cada arquivo em média teve uma redução do tempo de compilação de 52,3% com cada elemento da amostra tendo a redução entre o intervalo de 47,4% - 57,2% com confiança estatística de 95%.

Também constatou-se que, em média, uma linha de código em XIB é convertida para 0,94 linhas em Swift após o processo de conversão. Cada elemento da amostra apresenta uma proporção de conversão variando entre o intervalo de 0,91 - 0,97 linhas em Swift, com confiança estatística de 95%. Embora a conversão não altere significativamente a quantidade de linhas, promove-se a manutenibilidade pois troca-se XML por código Swift.

3.3 Avaliação da aplicabilidade da ferramenta

Nesta seção, é explorada a avaliação e validação da aplicabilidade da ferramenta em auxiliar o desenvolvedor na transição de projetos que utilizam a abordagem com o *Interface Builder* para a abordagem exclusivamente utilizando Swift.

3.3.1 Metodologia. Foi elaborado um questionário com oito perguntas, sendo cinco utilizando a escala Likert, as quais podem ser visualizadas na Figura 2, e três perguntas de resposta aberta: "Quais os principais problemas/dificuldades/obstáculos encontrados durante a conversão de forma manual?", "Quais os principais pontos positivos do xib2swift?" e "Quais os principais pontos de melhoria/evolução do xib2swift?". O questionário foi enviado a uma equipe de oito desenvolvedores que utilizaram a ferramenta para a conversão dos arquivos do referido sistema, o qual todos responderam anonimamente, de forma a incentivar respostas sinceras e sem viés. O foco das perguntas é verificar como é o processo de conversão manual, validar a aplicabilidade da ferramenta e identificar possíveis melhorias.

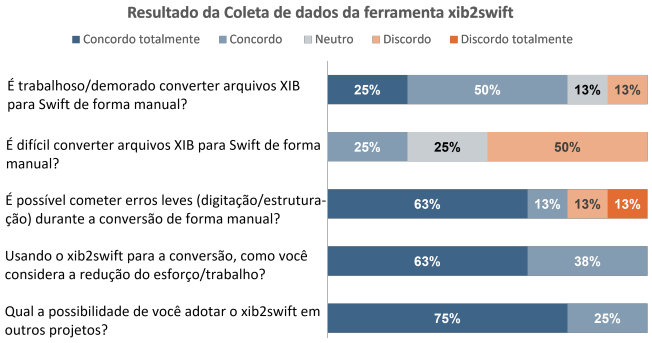

Figura 2: Respostas do questionário

3.3.2 Avaliação e Discussão. Analisando os resultados da Figura 2, é possível perceber que, por mais que o processo de conversão manual seja trabalhoso, não é difícil, porém está suscetível a erros. Todos concordaram que a ferramenta foi útil e eficaz na conversão e que a utilizariam em projetos futuros. É possível obter mais detalhes dos problemas da conversão manual nas respostas da pergunta "Quais os principais problemas/dificuldades/obstáculos encontrados durante a conversão de forma manual?", onde os trechos das seguintes de respostas foram encontrados:

> *"[...] além de trabalho e demorado, estava sujeito a falhas e crashes [...]" (dev 1), "[...] a análise através da visão XML do .xib pode gerar bastante confusão [...]" (dev 3), "[...] Ficar procurando todas as propriedades dentro do arquivo XML[...]" (dev 6), "Trabalho repetitivo, maçante e passível de erros." (dev 7), "Especialmente o tempo. É bastante trabalhoso e pode gerar erros." (dev 8).*

Também é possível evidenciar ainda mais a eficácia da ferramenta em reduzir o trabalho mecânico e aumentar a produtividade na pergunta "Quais os principais pontos positivos do xib2swift?", com as seguintes respostas:

> *"[...] agilidade, ele monta os itens com o nome que está no XML, com propriedades e constraints, deixa no formato/construção que utilizamos [...]" (dev 1), "[...] a maior parte da conversão já vem pronta [...]" (dev 2), "[...] elimina quase completamente o trabalho manual e mecânico do processo de conversão manual [...]" (dev 3), "ajuda muito no processo burocrático" (dev 4)", "[...] ele cria tudo e economiza um tempo bastante relevante para a produtividade [...]" (dev 5), "Agilidade e muito pouco ou quase zero retrabalho." (dev 8).*

Contudo, a ferramenta ainda precisa de ajustes e melhorias, como pode ser evidenciado nas respostas da pergunta "Quais os principais pontos de melhoria/evolução do xib2swift?":

> *"nomes de variáveis e soltar o código direto no Xcode (editando no arquivo .swift)" (dev 2), "[...] não incluir configurações redundantes (geradas por valores default que são explicitamente declarados no XML do .xib [...]" (dev 3), "Ele exclui poucas propriedades que vem como padrão, temos que apagar manualmente" (dev 6), "[...] Melhorar nomeação de elementos não nomeados [...]" (dev 7).*

Todas as respostas do questionário podem ser encontradas por completo na página do projeto dentro da pasta *survey*.[4]

4 CONSIDERAÇÕES FINAIS

Diante do cenário em constante evolução do desenvolvimento iOS e a necessidade crescente de manter as práticas de desenvolvimento atualizadas e utilizar abordagens que trazem ganhos a projetos antigos, a ferramenta apresentada neste artigo representa uma contribuição valiosa para a comunidade de desenvolvedores iOS, já que ela agiliza e auxilia o processo de conversão das abordagens que utilizam o *Interface Builder* para a abordagem adotada pelo mercado que utiliza apenas Swift.

Espera-se que essa ferramenta seja adotada e aprimorada por desenvolvedores e empresas em processos de migração de código.

Além disso, é sugerido que futuros trabalhos explorem ainda mais as possibilidades de conversão, fazendo uma versão de conversão para o novo *framework* de desenvolvimento, o SwiftUI. É possível também ser explorado a contínua colaboração dentro de um projeto de código aberto, o qual necessita acompanhar o ritmo das mudanças das novas versões do *Interface Builder* para garantir o funcionamento da ferramenta.

[4]Disponível em: https://github.com/vinicius-caputo/xib2swift/tree/main/survey

REFERÊNCIAS

[1] Pedro Alvarez. 2022. Storyboards x Xibs x ViewCode: Which one is better? Disponível em: <https://pedroalvarez-29395.medium.com/storyboards-x-xibs-x-viewcode-which-one-is-better-6e0c7a24b867>. Acesso em: 30 abr. 2024.

[2] Gary Bennett, Mitch Fisher, and Brad Lees. 2010. *Objective-C for absolute beginners: iPhone, iPad, and Mac programming made easy.* Apress, Berkeley, Estados Unidos.

[3] Donny Clayton, Craig; Wals. 2019. *Complete IOS 12 Development Guide: Become a Professional IOS Developer by Mastering Swift, Xcode 10, ARKit, and Core ML.* Packt Publishing Ltd, Birmingham, Inglaterra.

[4] M. Dippery. 2015. *Professional Swift.* John Wiley & Sons, Indianapolis, Estados Unidos.

[5] Craig Grummitt. 2017. *iOS Development with Swift.* Simon and Schuster, Shelter Island, Estados Unidos.

[6] Jack; LaMarche Jeff Mark, Dave; Nutting. 2011. *Beginning iOS 5 Development: Exploring the iOS SDK.* Apress, Berkeley, Estados Unidos.

[7] Giovanna Moeller. 2022. iOS e Swift: Diferenças na construção de layouts com Storyboard, XIB e View Code. Disponível em: <https://www.alura.com.br/artigos/ios-swift-diferencas-construcao-layouts-storyboard-xib-view-code>. Acesso em: 17 set. 2023.

[8] Alexander Nekrasov. 2022. *UIKit and Storyboards.* Apress, Berkeley, Estados Unidos.

[9] Donny Wals. 2017. *Mastering iOS 11 Programming* (2nd ed.). Packt Publishing Ltd, Birmingham, Inglaterra.

[10] Richard Wentk. 2010. *Cocoa.* John Wiley & Sons, Indianapolis, Estados Unidos.