



Full Stack

Node.js

Git e Github

DigitalHouse>

O que vimos no Playground

- Introdução ao GIT
- Instalação do GIT
- Github
- Criando seu primeiro repositório
- Confirmando arquivos
- Adicionando arquivos
- Guia prático



O que vamos ver hoje

- Versionamento de código local e na nuvem
- Código colaborativo com segurança



GIT E **GITHUB**, SÃO
INDEPENDENTES ENTRE SI?

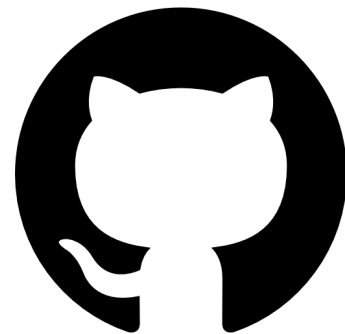
GIT

- Controlador de versão local:
 - autor,
 - data e hora,
 - histórico do projeto recuperável
- É utilizado em várias plataformas online como GitHub, GitLab e BitBuket



GITHUB

- Armazenar código (mais seguro do que salvar apenas no computador)
- Portfólio online (vitrine de aplicações e desafios realizados em programação)
- Código colaborativo (mais de um participante do projeto - empresas utilizam)
- Open source (código público para toda a comunidade)



Revisando comandos essenciais

```
git remote add origin url
```

```
git clone url-do-repositorio
```

```
git add caminho-do-arquivo
```

```
git push origin master
```

adicionando url do repositório online

clone na máquina local do repositório

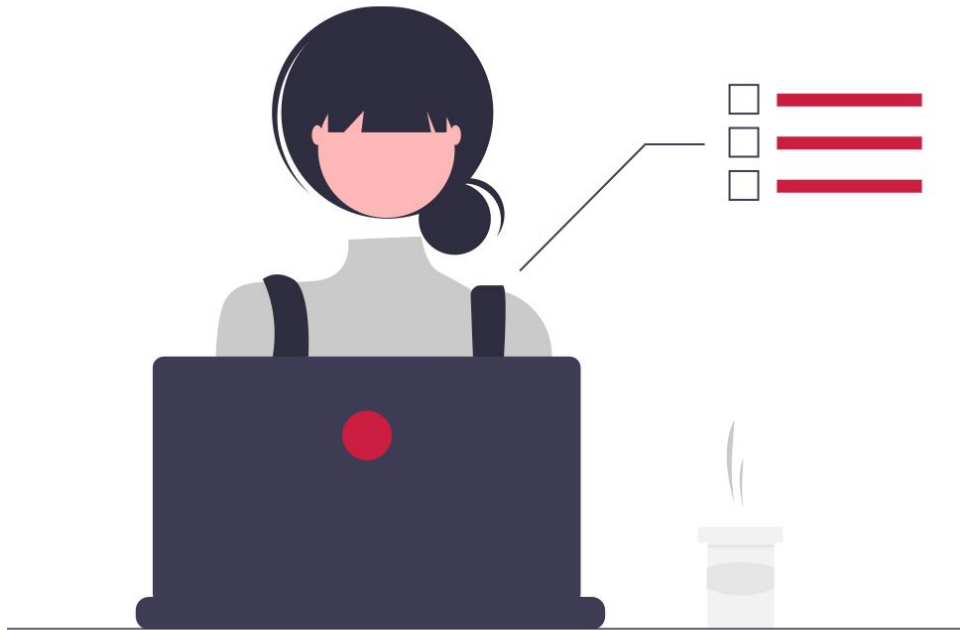
incluir arquivos no próximo commit

enviar atualizações para repositório

Revisando comandos essenciais

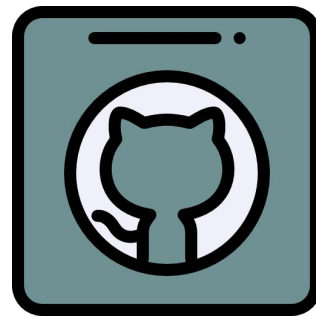
<code>git pull <u>origin</u> <u>master</u></code>	trazer atualizações do repositório
<code>git commit -m "mensagem"</code>	salvando versão do código
<code>git status</code>	status do repositório local
<code>git log</code>	visualizar histórico de modificações

Hora de praticar!



Atividade I - Nosso projeto no github

- Inicie o versionamento no projeto **x**
- Crie um repositório online para o projeto com o mesmo título que utilizou em seu computador
- Crie um commit com todos os arquivos e envie o projeto para o repositório no GitHub recém-criado

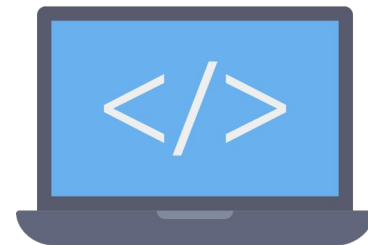


Hora de praticar!



Atividade II - Um projeto colaborativo

- Em duplas/trios escolham quais alunos serão o A, B e C
- Ver documento com as instruções detalhadas que cada membro da equipe deve cumprir



Guia de Instalação Git

DigitalHouse >
Coding School

Índice

Linux

Mac

Windows

1 | Linux

Git

O que é o Git?

Como fazer a instalação?

“**Git** é um software de controle de versão desenvolvido por Linus Torvalds.”



Como instalá-lo

No console, execute:

```
>_ sudo apt get update
```

```
>_ sudo apt install git
```

2 | Mac

Git / GitKraken

Como fazer a instalação?

O que é o GitKraken?

Como fazer a instalação?

Como instalá-lo

Siga as instruções disponíveis na página oficial do Git, na seção “[Instalando no Mac](#)”. Ao concluir, use o comando abaixo para confirmar que está tudo instalado:

```
>_ git --version
```

GitKraken

“ É uma interface gráfica para utilizar o **git**. O **GitKraken** **não** é mais produtivo que o **Git Bash** quando se trata de comandos rotineiros. ”



Como fazer a instalação?

1. Visite o site <https://www.gitkraken.com/download>

1. Clique no botão download referente a sua plataforma:



1. Siga os passos descritos no vídeo de instalação, na própria página

3 | Windows

Git/ Git Bash

O que é o GitBash?

Como fazer a instalação?

“ **Git Bash** é um terminal que nos permite usar o **git** no Windows com mais facilidade. ”



Como fazer a instalação?

1. Visite o site <https://git-scm.com/downloads>

1. Clique link para fazer o download:



1. Siga as instruções do instalador. Nas telas seguintes, as opções que já vem marcadas funcionam muito bem. Somente as altere se souber o que está fazendo.

GitKraken

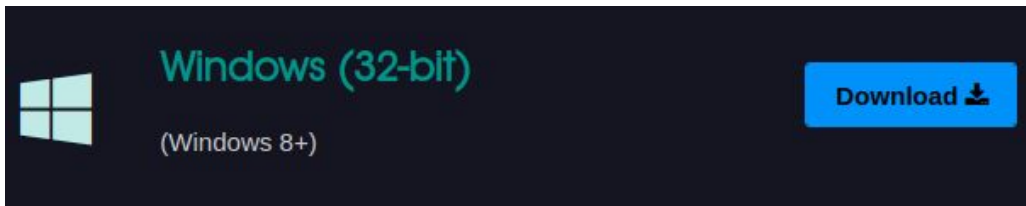
O que é o GitKraken?

Como fazer a instalação?

Como fazer a instalação?

1. Visite o site <https://www.gitkraken.com/download>

1. Clique no botão download referente a sua plataforma:



1. Abra o arquivo baixado e siga as instruções de instalação.



Repositório Git

“ Os seguintes **comandos** são o **passo a passo** que necessitamos para **trabalhar** com um **repositório**. ”



Comandos passo a passo

```
>_ git init // criar repositório
```

```
>_ git config user.name "nomeUsuario" // criar repositório
```

```
>_ git config user.email "emailUsuario" // criar repositório
```

```
>_ git remote add origin https://...// aponta o repositório  
remoto
```

Comandos passo a passo

```
>_ git add . // adiciona todas as modificações
```

```
>_ git commit -m "mensagem" // commita(salva) as  
modificações feitas
```

```
>_ git push origin master // envia as modificações ao  
repositório remoto
```

Lembre-se de que é sempre bom ter a **documentação** à mão, para isso podemos visitar:

<https://dev.to/t/git/>

<https://ohshitgit.com>

<https://git-scm.com>



Confirmando arquivos

“ Os seguintes **comandos** são o **passo** a **passo** que necessitamos para **confirmar que nossos arquivos** se encontram em nosso **repositório**. ”



Comandos passo a passo

É necessário enfatizar que, para criar um commit, você precisa seguir os seguintes passos:

```
>_ git status //deve ser feito antes e depois de adicionar  
os adicionar os arquivos e realizar os commits
```

```
>_ git add . // adiciona todas as modificações
```

```
>_ git commit -m "mensagem" // comita (salva) as  
modificações feitas
```

Sintaxe

Comando

O commit é a confirmação ao repositório de que os arquivos foram adicionados.

Mensagem

A mensagem serve como um guia do que foi adicionado ou alterado no commit, para que fique claro para aqueles que vejam o projeto.



```
>_ git commit -m "mensagem"
```

The diagram shows the command `git commit -m "mensagem"` in a terminal window. Above the command, there are two curly braces: an orange one spanning `commit -m` and a grey one spanning `"mensagem"`. Below the command, a teal curly brace spans the entire command `git commit -m "mensagem"`.

Objetivo

Os commits geram uma parte cronológica na linha do tempo do projeto, possibilitando revisitá-los, caso seja necessário.

Lembre-se de que é sempre bom ter a **documentação** à mão, para isso podemos visitar:

<https://dev.to/t/git/>

<https://ohshitgit.com>

<https://git-scm.com>



Adicionando archivos

“ Os seguintes **comandos** são o **passo** a **passo** que necessitamos para **adicionar arquivos** em um **repositório local**. ”



Comandos passo a passo

```
>_ git status // para certificar quais arquivos precisamos  
adicionar
```

```
>_ git add nome_arquivo // adicionar um arquivo específico
```

```
>_ git add . // adicionar todos os arquivos
```

Lembre-se de que é sempre bom ter a **documentação** à mão, para isso podemos visitar:

<https://dev.to/t/git/>

<https://ohshitgit.com>

<https://git-scm.com>



Subindo arquivos

“ Os seguintes **comandos** são o **passo a passo** que necessitamos para **subir nossos arquivos** em nosso **repositório remoto no github**. ”



Comandos passo a passo

```
>_ git push origin master // envia as modificações ao  
repositório remoto
```

```
>_ git push // para adicionar todos os arquivos ao  
repositório remoto
```

Sintaxe de push

Comando

Sinaliza que você quer enviar os arquivos que possui no repositório local para o repositório remoto.

Branch

A branch indica a localização do seu arquivo no repositório, já que podemos ter várias branches. Neste caso, a master é a branch principal do nosso repositório.



```
>_ git push origin master
```

Nome do repositório

Para o git, o repositório remoto é chamado de origin, isso indica o local onde você está enviando o seu arquivo.

Lembre-se de que é sempre bom ter a **documentação** à mão, para isso podemos visitar:

<https://dev.to/t/git/>

<https://ohshitgit.com>

<https://git-scm.com>



Baixando arquivos

“ Os seguintes **comandos** são o **passo a passo** que necessitamos para **baixar a última versão dos arquivos** de um **repositório remoto**. ”



Comandos passo a passo

```
>_ git clone https://github.com/meunome/arquivo-git.git
```

```
>_ git pull origin master // vai baixar e atualizar apenas  
os arquivos que sofreram atualização
```

Sintaxe

Indica qual é a url do repositório remoto de onde os arquivos estão publicados.



```
>_ git clone https://github.com/meunome/arquivo-git.git
```

Cria uma cópia exata na máquina de todos os arquivos existentes em um repositório remoto.

Lembre-se de que é sempre bom ter a **documentação** à mão, para isso podemos visitar:

<https://dev.to/t/git/>

<https://ohshitgit.com>

<https://git-scm.com>



Git

Avançado

Índice

1. Branch
2. GIT LOG
3. Head
4. Desfazendo Mudanças
5. Unindo Branches
6. Conflitos

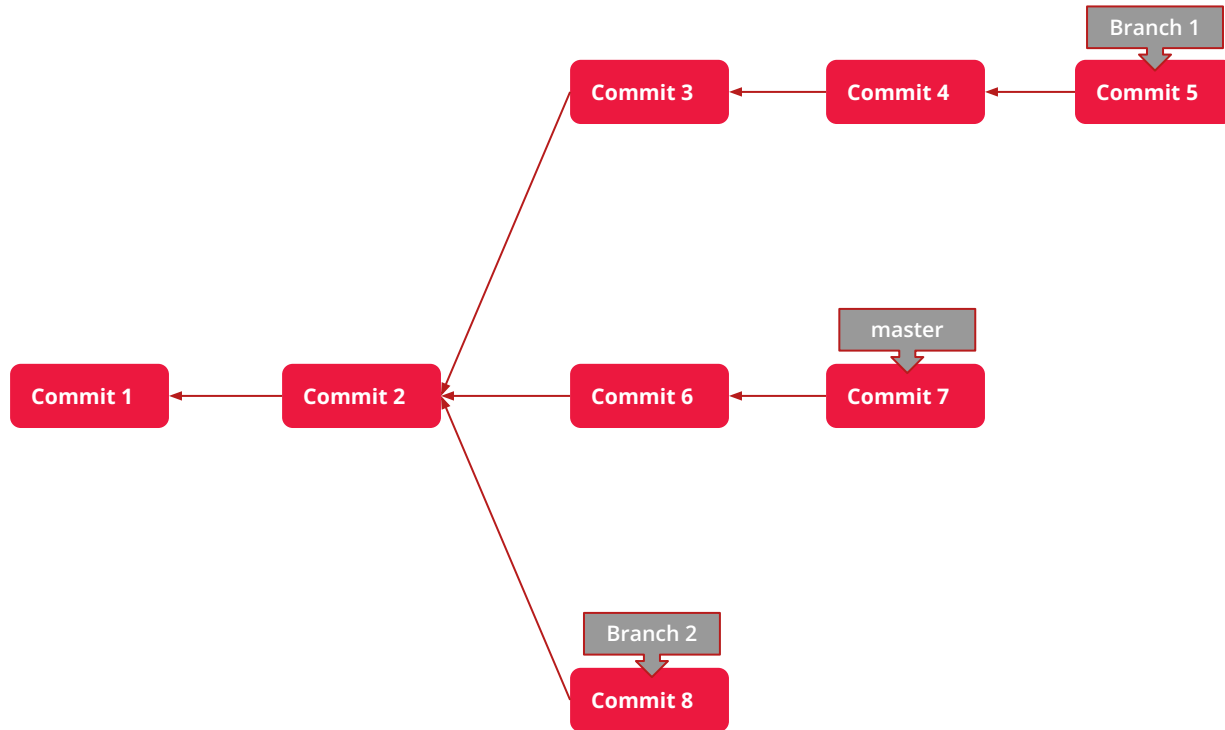
1



Branches

“ São uma ‘linha’ de desenvolvimento diferente da utilizada para testar novas funcionalidades sem interferir na linha principal, chamada de **master**. ”

BRANCHES



Criando Branches

Para criar uma branch de nome "bugfix45".

```
>_ git branch bugfix45
```

Para passar a utilizar a branch "bugfix45".

```
>_ git checkout bugfix45
```

Criando Branches

Para criar uma branch de nome “bugfix45” e passar a utilizá-la com um só comando.

```
>_
```

```
git checkout -b bugfix45
```

Listando Branches

Para listar todas as branches de um repositório local execute.

```
>_ git branch
```

Para listar todas as branches locais e remotas execute.

```
>_ git branch -a
```

2



GIT LOG

GIT LOG

Podemos listar o histórico de commits utilizados com o comando **git log**. Este comando exibe todos os commits.

```
>_ git log
```

Entre outras coisas, esse comando exibe o nome do autor, data e hora da realização do commit e o **código hash** que identifica a cada commit.

```
>_ commit 268a7cc0b402bfd4bf89ae61ea743e8c1c6f33f3 (HEAD -> master)
    Author: Bob Esponja <spongebob@bikinibottom.com>
    Date:   Wed Jul 22 00:45:01 2020 -0300
```


3 | **Head**

“ O **HEAD** é uma referência ao estado atual do seu repositório. É comum que ele esteja apontando para o último commit. ”

Head



GIT CHECKOUT

Podemos utilizar o comando `git checkout` para mover a HEAD do repositório para qualquer branch.

```
>_ git checkout bugfix45
```



GIT CHECKOUT

Podemos utilizar o comando `git checkout` para mover a HEAD para qualquer commit. Basta saber os 6 primeiros caracteres do **hash** do commit.

```
>_ git checkout a6de24
```



Para saber mais

Alguns conceitos do uso de branch, checkout entre outras coisas, jogue em:

https://learngitbranching.js.org/?locale=pt_BR

Um guia ilustrado do comportamento do projeto usando GATINHOS!

<https://girliemac.com/blog/2017/12/26/git-purr/>



4

Desfazendo Mudanças

Desfazendo

Para desfazer as mudanças que não foram adicionadas ao stage.

```
>_ git checkout -- <nomeDoArquivo>
```

Para desfazer as mudanças dos arquivos no stage, mas não commitadas.

```
>_ git checkout HEAD -- <nomeDoArquivo>
```



Executando esses comandos, o arquivo volta para a sua versão do último commit

GIT REVERT

Cria um novo commit revertendo as alterações feitas até o commit especificado.

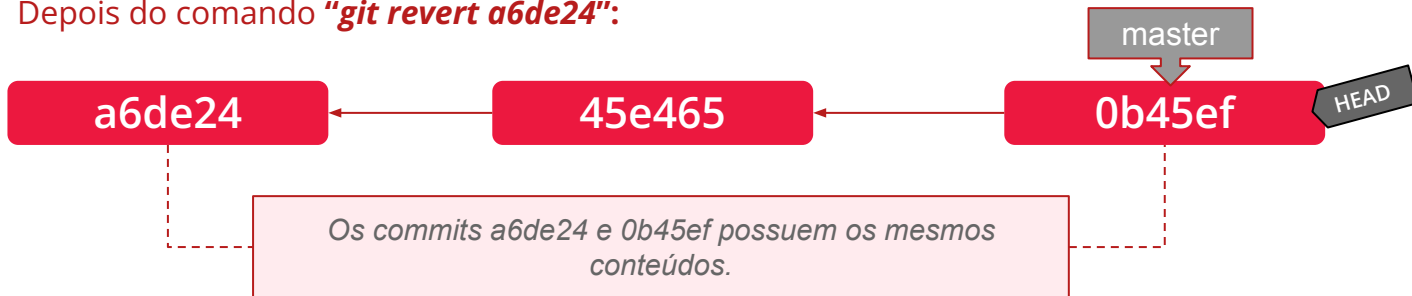
```
>_
```

```
git revert <HASH-DE-COMMIT-ANTERIOR>
```

Antes:



Depois do comando ***"git revert a6de24"***:



GIT

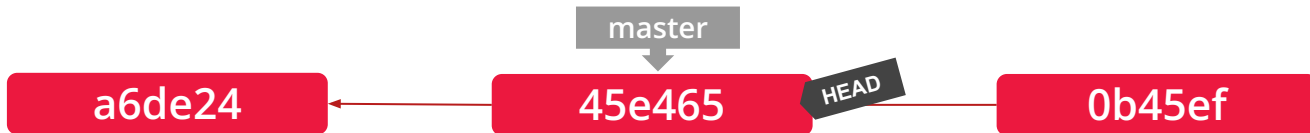
Move o apontador da branch para um outro commit.

```
>_ git reset --hard <HASH-DE-COMMIT>
```

Antes:



Depois do comando “`git reset --hard 45e465`”:



Branch master passa a apontar para o commit 45e465. O commit 0b45ef continua existindo. Para ver ele e outros, utilize o comando “`git reflog`”.

5

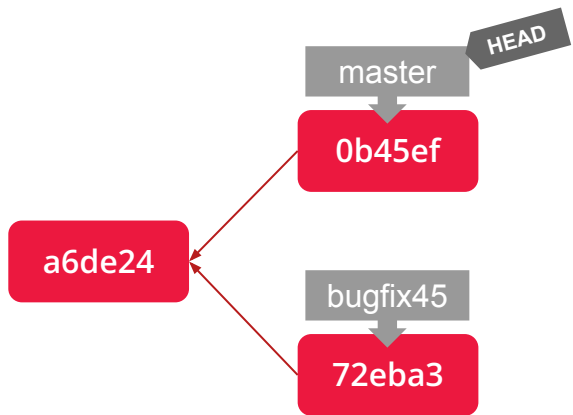
Unindo Branches

GIT

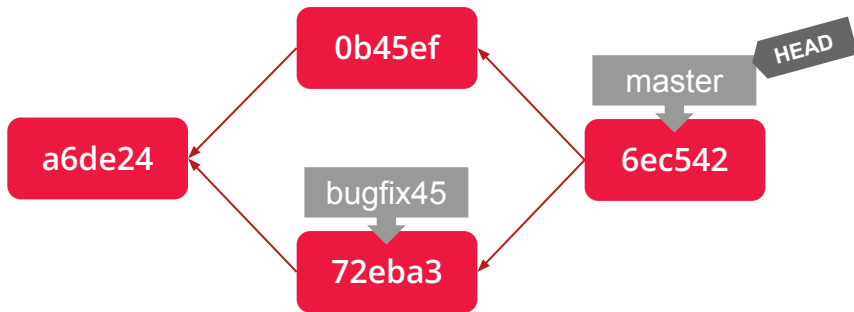
Tenta criar um novo commit unindo a branch atual a uma outra branch.

```
>_ git merge <OUTRA_BRANCH>
```

Antes:



Depois do comando "git merge bugfix45"

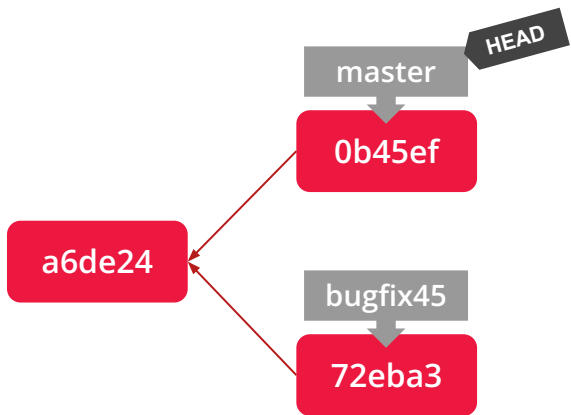


GIT

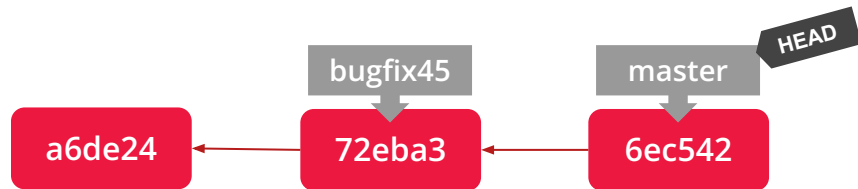
Emenda o branch atual sobre a outra branch.

```
>_ git rebase <OUTRA_BRANCH>
```

Antes:



Depois do comando **"git rebase bugfix45"**



O primeiro commit comum às branches `master` e `bugfix45` é o `a6de24`. Inicialmente, ele é o commit base. Depois de executado o comando a base do branch `master` e passa a ser o commit `72eba3`, que é a ponta do branch `bugfix45`.

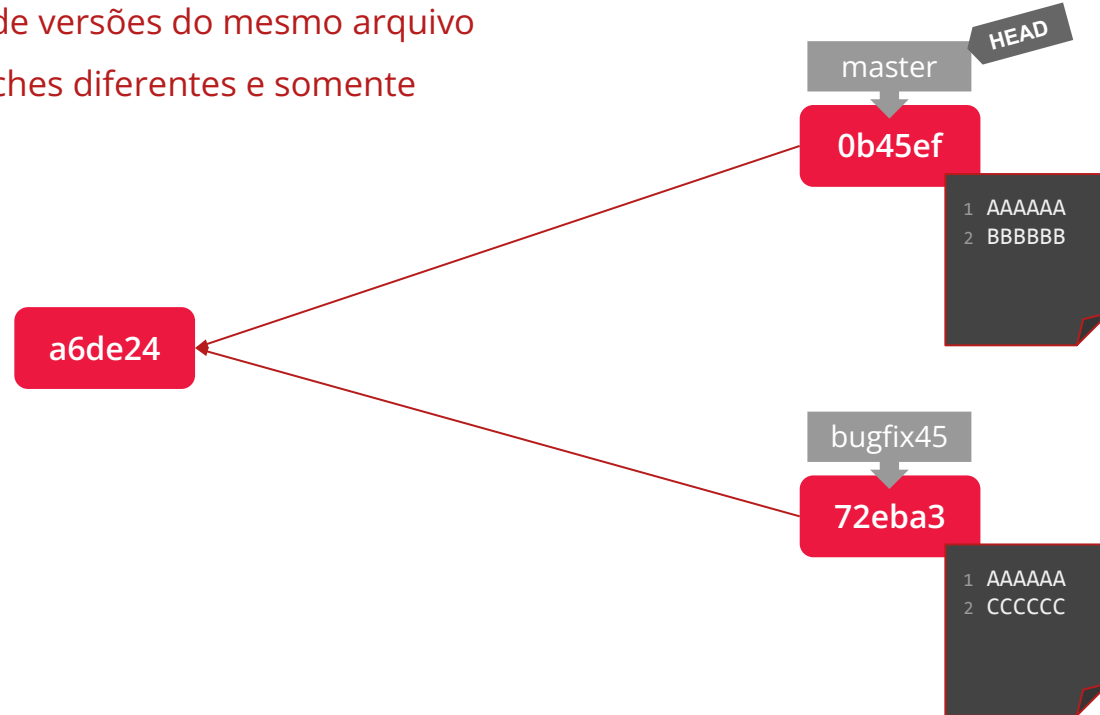
6

Conflitos

“ As operações de **união de branches** podem provocar **conflitos**: ocorre quando o **mesmo arquivo** tem a **mesma parte diferente** nas duas branches. ”

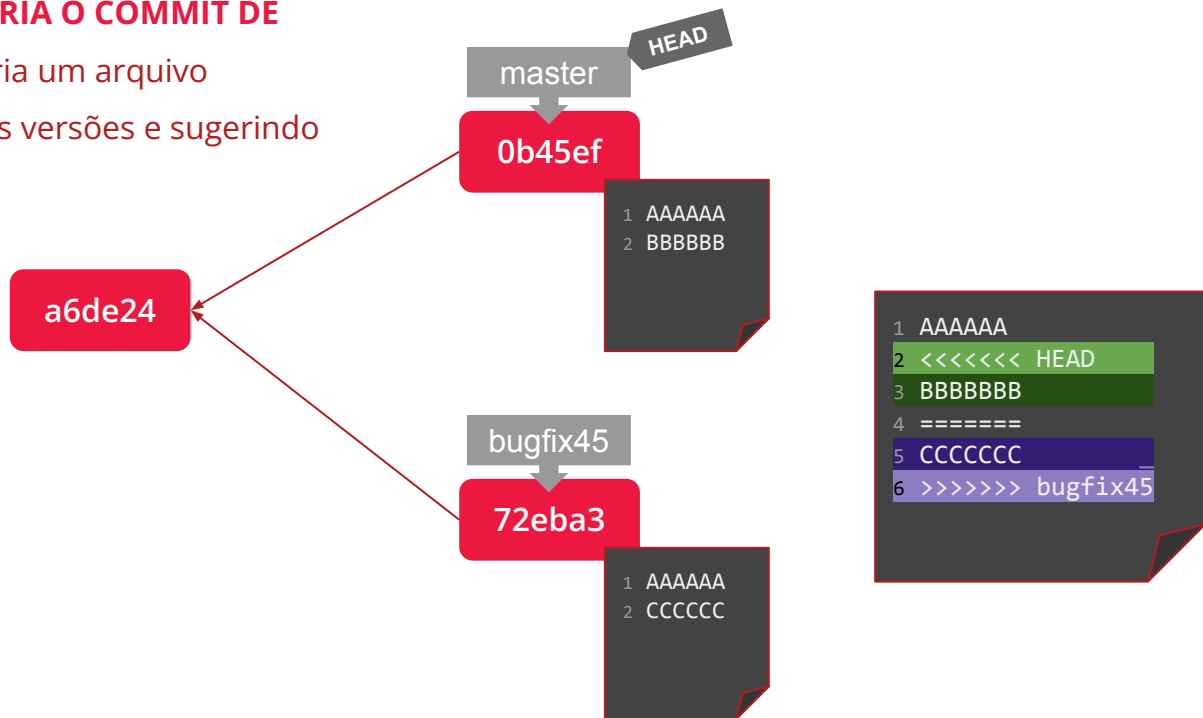
Resolução de CONFLITOS

Observe a situação onde versões do mesmo arquivo existem em duas branches diferentes e somente linha 2 tem diferença.



Resolução de CONFLITOS

Neste caso, quando executamos o comando "git merge bugfix45" o git **NÃO CRIA O COMMIT DE UNIÃO**. Ao invés disso, ele cria um arquivo mostrando o conflito entre as versões e sugerindo uma solução:

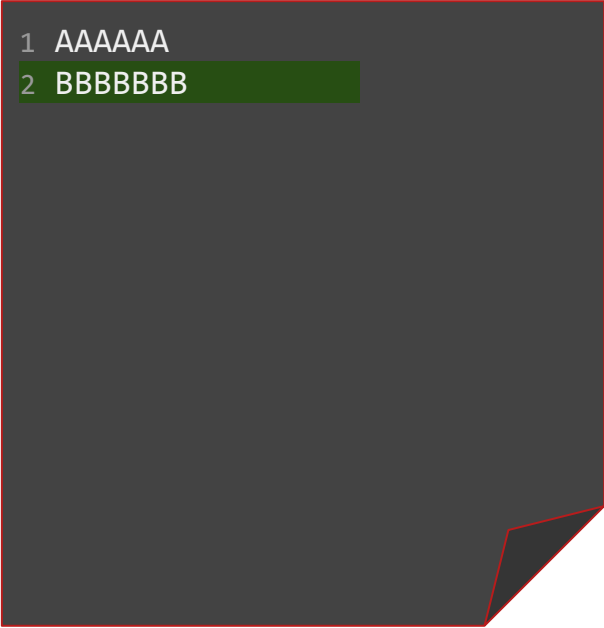


Resolução de CONFLITOS

```
1 AAAAAA
2 <<<<<< HEAD
3 BBBBBBB
4 =====
5 CCCCCC
6 >>>>>> bugfix45
```

Neste exemplo, as linhas 2, 4 e 6 foram adicionadas pelo git depois de executar o comando “git merge bugfix45”. Elas indicam onde o programador deve atuar para resolver o conflito. De modo geral, temos 3 opções.

Resolução de CONFLITOS

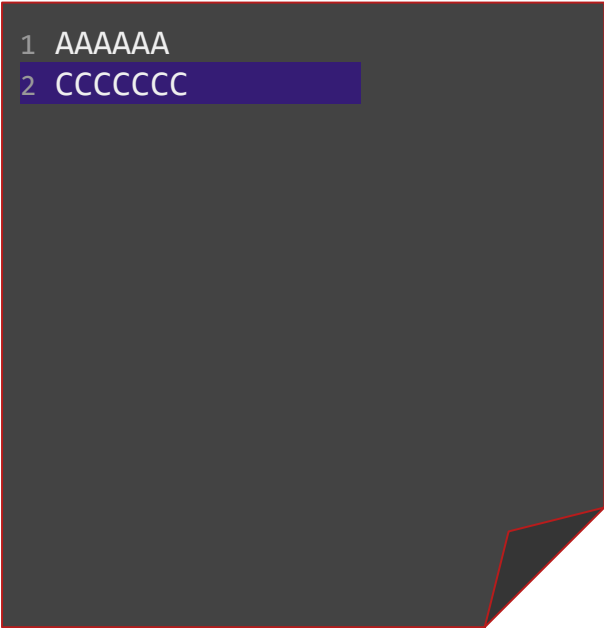


1 AAAAAA
2 BBBB BB

Opção 1:

Manter as linhas do branch atual (HEAD).

Resolução de CONFLITOS

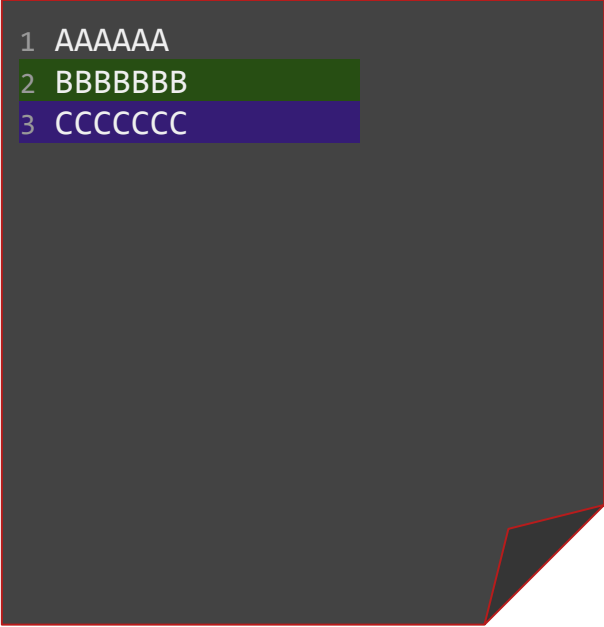


```
1 AAAAAA  
2 CCCCCC
```

Opção 2:

Manter as linhas do outro branch (bugfix45).

Resolução de CONFLITOS



```
1 AAAAAA  
2 BBBB BB  
3 CCCCCC
```

Opção 3:

Manter todas as linhas.

Resolução de **CONFLITOS**

```
1 AAAAAA
2 if(x > y) {
3    BBBBBB
4 } else {
5     CCCCCC
6 }
```

Em alguns casos, o programador deverá alterar/complementar o código para garantir o funcionamento geral do sistema.

Resolução de **CONFLITOS**

Qualquer que seja a decisão tomada, o conflito somente será resolvido depois que todo o arquivo com conflito for adicionado ao stage.

```
>_
```

```
git add arquivoDoConflito.js
```

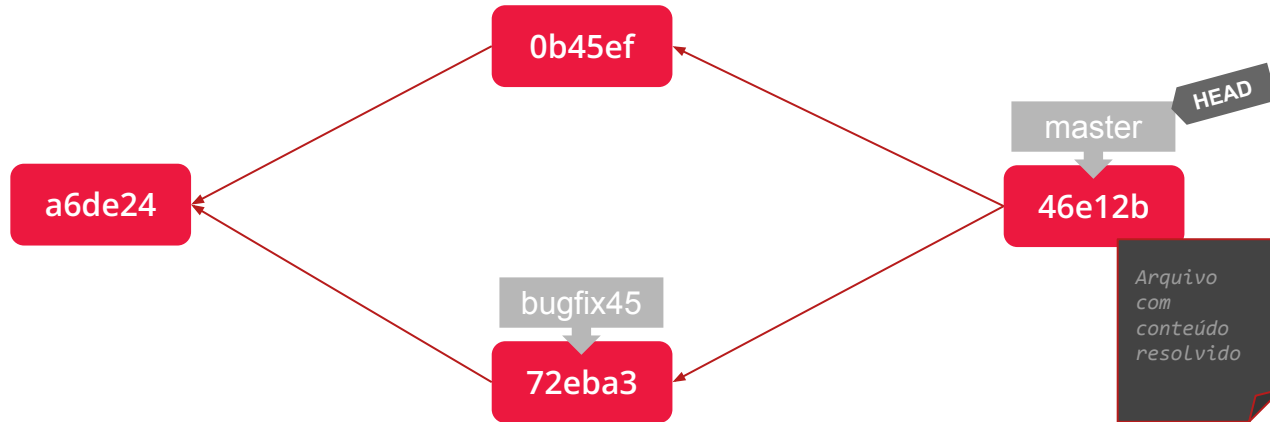
O commit que une os dois branches somente será criado quando o programador executar o comando que o cria.

```
>_
```

```
git commit -m "Conflito resolvido com sucesso!"
```

Resolução de **CONFLITOS**

Ao final do processo, o resultado esperado deve ser este:



Comandos Git

Comandos GIT

→ Ajuda

- ❑ `git help -> comando geral`

→ Comando específico

- ❑ `git help add`
- ❑ `git help commit`
- ❑ `git help <qualquer_comando_git>`

→ Setar usuário e email

- ❑ `git config --global user.name "nome do usuário"`
- ❑ `git config --global user.email email@email.com`

→ Remover todas as linhas que referenciam o usuário e email

- ❑ `git config --global --unset user.name "nome do usuário"`
- ❑ `git config --global --unset user.email email@email.com`

→ Lista configurações

- ❑ `git config --list`

→ Criando novo repositório

- ❑ `git init`

Comandos GIT

→ Verificar o estado dos arquivos/diretórios

- ❑ `git status` (mostra qual a situação do arquivos no seu repositório)

→ Adicionando arquivo

- ❑ `git add meu_diretorio` (arquivo específico)
- ❑ `git add .` / `git add --all` (todos os arquivos)

→ Comitar arquivo/diretório

- ❑ `git commit arquivo -m "mensagem de commit"`

→ Remover arquivo/diretório

- ❑ `git rm arquivo` (remove arquivo)
- ❑ `git rm -r diretório` (remove diretório/pasta)

→ Visualizar histórico

- ❑ `git log` (exibir histórico)
- ❑ `git log -- <caminho_do_arquivo>` (exibir histórico de um arquivo específico)
- ❑ `git log --author=usuário` (exibir histórico de um determinado)

Desfazendo Operações

→ Desfazendo alteração local no seu diretório de trabalho local

- ❑ `git checkout -- arquivo` (Este comando só deve ser utilizado enquanto o arquivo ainda não foi adicionado na área de trabalho temporária)

→ Desfazendo alteração local na área de trabalho temporária(staged área)

- ❑ `git reset HEAD arquivo` (Este comando deve ser utilizado quando o arquivo já foi adicionado na área temporária)
- ❑ `Unstaged changes after reset:M` arquivo (se o resultado abaixo for exibido, o comando `reset` não alterou o diretório de trabalho)
- ❑ `git checkout arquivo` (a alteração do diretório pode ser realizada através deste comando)

Repositório Remoto

- Exibir os repositórios remotos (Para sabermos para onde estão sendo enviadas nossas alterações ou de onde estamos baixando as coisas)
 - ❑ `git remote`
 - ❑ `git remote -v`
 - ❑ `git remote add origin git@github.com:meunome/arquivo-git.git` (vincular repositório local com um repositório remoto)
 - ❑ `git remote show origin` (exibir informações dos repositórios remotos)
 - ❑ `git remote rename origin arquivo-git` (renomear um repositório remoto)
 - ❑ `git remote rm arquivo-git` (desvincular um repositório remoto)
 - ❑ `git push -u origin master` (o primeiro push no repositório deve conter o nome do mesmo e a sua branch)
 - ❑ `git push` (os demais pushes não precisam de outras informações)

- Atualizar repositório local de acordo com o repositório remoto
 - ❑ `git pull` (atualizar os arquivos em relação a branch atual)
 - ❑ `git fetch` (buscar as alterações, mas não aplicá-las na branch atual)

- Clonar um repositório remoto já existente
 - ❑ `git clone git@github.com:meunome/arquivo-git.git`

Branches

A master é a branch principal do GIT.

O **HEAD** é um ponteiro especial que indica qual é o branch atual. Por padrão, o HEAD aponta para o branch principal, o master.

- ❑ `git branch novaBranch_nome` (criando uma nova branch)
- ❑ `git checkout novaBranch_nome` (trocando para uma branch existente). Neste caso, o ponteiro principal HEAD esta apontando para o branch chamada novaBranch_nome.
- ❑ `git checkout -b novaBranch_nome` (cria uma nova branch e troca pra ela)
- ❑ `git checkout master` (voltar para a branch principal(master))
- ❑ `git merge novaBranch_nome` (resolve o merge entre duas branches) - para realizar o merge, é necessário estar no branch que deverá receber as alterações.
- ❑ `git branch -d novaBranch_nome` (apagando uma branch)
- ❑ `git branch` (listar branches)
- ❑ `git branch -v` (listar branches com informações dos últimos commits)

Branches

- ❑ `git branch --merged` (listar branches já foram fundidos(merged) com a master)
- ❑ `git branch --no-merged` (listar branches que não foram fundidos(merged) com a master)
- ❑ `git pull origin nomeBranch` (puxando arquivos de uma branch já existente)
- ❑ `git push origin novaBranch_nome` (criando um branch remoto com o mesmo nome)
- ❑ `git merge --abort` ou `git reset --merge` (quando você tem problemas com merge e deseja desfazer o merge)
- ❑ `git reset HEAD` (quando você deseja voltar commits, caso você queira voltar mais de um commit, coloque o número dos commits após o HEAD -> exemplo HEAD~2)

→ Reescrevendo o histórico

- ❑ `git commit --amend -m "Minha nova mensagem"` (alterando mensagens do commit)

Comandos para o Terminal

→ **crtl + l ou clear**

☐ limpa tela

→ **mkdir nome_da_pasta**

☐ cria uma pasta

→ **cd**

☐ entrar na pasta

→ **cd ..**

☐ sair da pasta

→ **ls**

☐ ver o que tem dentro da pasta

→ **rm**

☐ deleta arquivo

→ **rm -r nome**

☐ remover

→ **rm -rf nome**

☐ remove tudo

SCRUM

O que vimos no playground

- Antes da Metodologia Ágil
- SCRUM



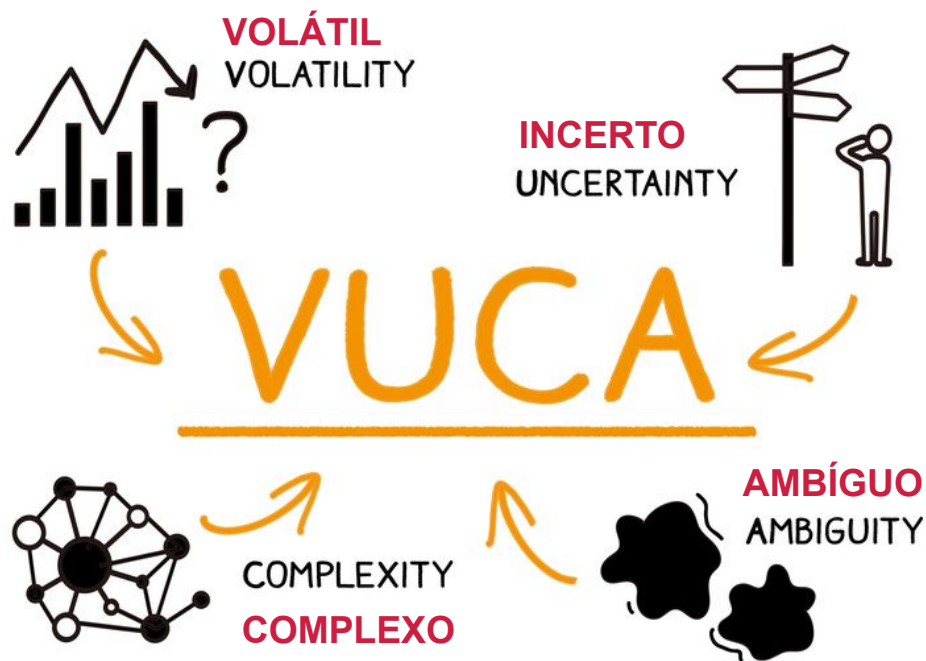
O que vamos ver hoje

- O que são metodologias ágeis
- Como Scrum pode ser aplicado em times de desenvolvimento
- Scrum no Projeto Integrador



QUAL A IMPORTÂNCIA DA
UTILIZAÇÃO DE **METODOLOGIAS**
ÁGEIS EM PROJETOS?

Cenário atual: mundo V.U.C.A



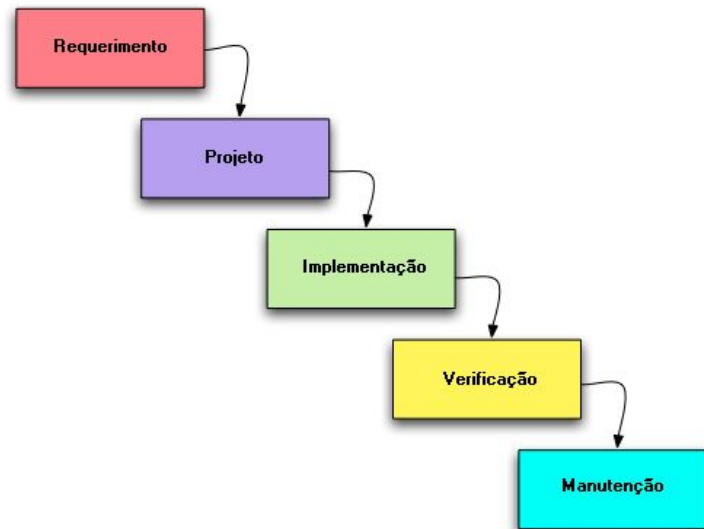
Projetos

- Esforço a cumprir com um produto, serviço ou resultado
- Tem começo, meio e fim
- Demanda recursos



Desenvolvimento de projetos sem implementação do AGILE: cascata (*waterfall*)

- Levantamento dos requisitos no início (e apenas neste momento)
- Entregas “engessadas” com a definição inicial
- Pouca possibilidade de mudanças



Desenvolvimento de projetos utilizando AGILE:

Funcionalidade A

Planejamento
Projeto / Design
Implementação / código
Verificação / Testes
Manutenção

Funcionalidade B

Planejamento
Projeto / Design
Implementação / código
Verificação / Testes
Manutenção

Funcionalidade C

Planejamento
Projeto / Design
Implementação / código
Verificação / Testes
Manutenção

=

Visualização do processo

?

Errar rápido e acertar rápido

AGILE combate à:

- Desenvolvimentos desnecessários
- Entregas demoradas
- Dificuldade na adaptação à mudanças
- Alta complexidade de tarefas (divisões de escopo)



0 FRAMEWORK

SCRUM

Pilares e objetivos:

Transparência

Inspeção

Comprometimento

Adaptação

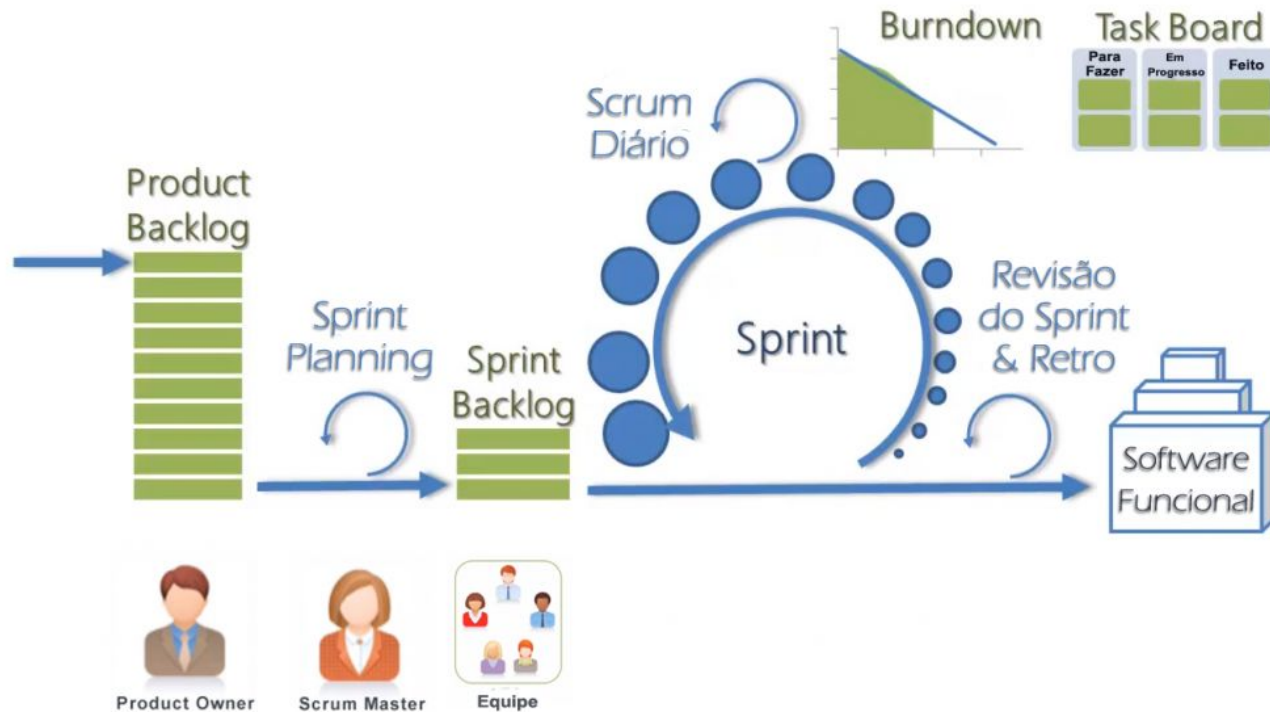
Foco

Sinceridade

Coragem

Respeito

Fluxo de trabalho Scrum



TIME SCRUM

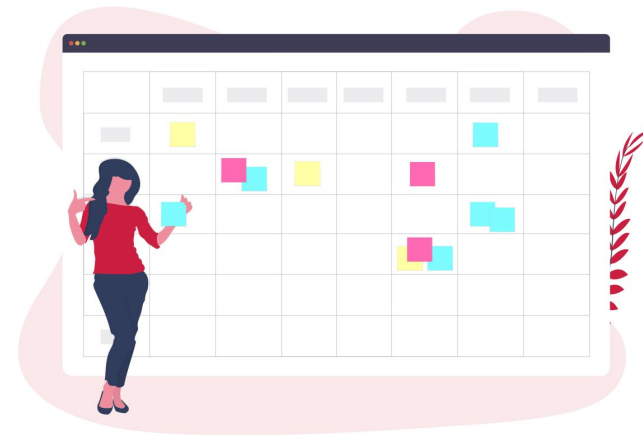
Scrum Master

- Orienta o time com relação as práticas Ágeis
- Remove impedimentos
- Busca inovações na organização de trabalho do time
- Ajuda com tirando dúvidas



Product Owner

- É o dono do backlog do produto
- Disponível ao time - blinda as demandas no backlog
- Alinha as expectativas do projeto
- Prioriza histórias e tarefas para sucesso do projeto e entregas



Dev Team

- Possui autonomia para distribuição de tarefas do backlog (transparência)
- Acertos e erros pertencem ao time - coletivo
- Responsáveis pelo desenvolvimento do projeto



Um pouco de história

“

Quando falamos em programação, uma das primeiras coisas que nos vem à mente é a automação de tarefas ou processos, afinal programamos porque queremos que o computador faça o trabalho por nós.

”



O que acontecia antes das metodologias ágeis?

A programação de tarefas, no sentido amplo da palavra, tem suas origens na **revolução industrial** de 1760 e continua em ritmo acelerado e sem pausa até os dias atuais.

As primeiras **metodologias de desenvolvimento** de software se inspiraram nesses processos industriais e propuseram um **modelo sequencial de etapas**, em que o início de uma etapa do projeto depende exclusivamente do encerramento da etapa anterior.

O que acontecia antes das metodologias ágeis?

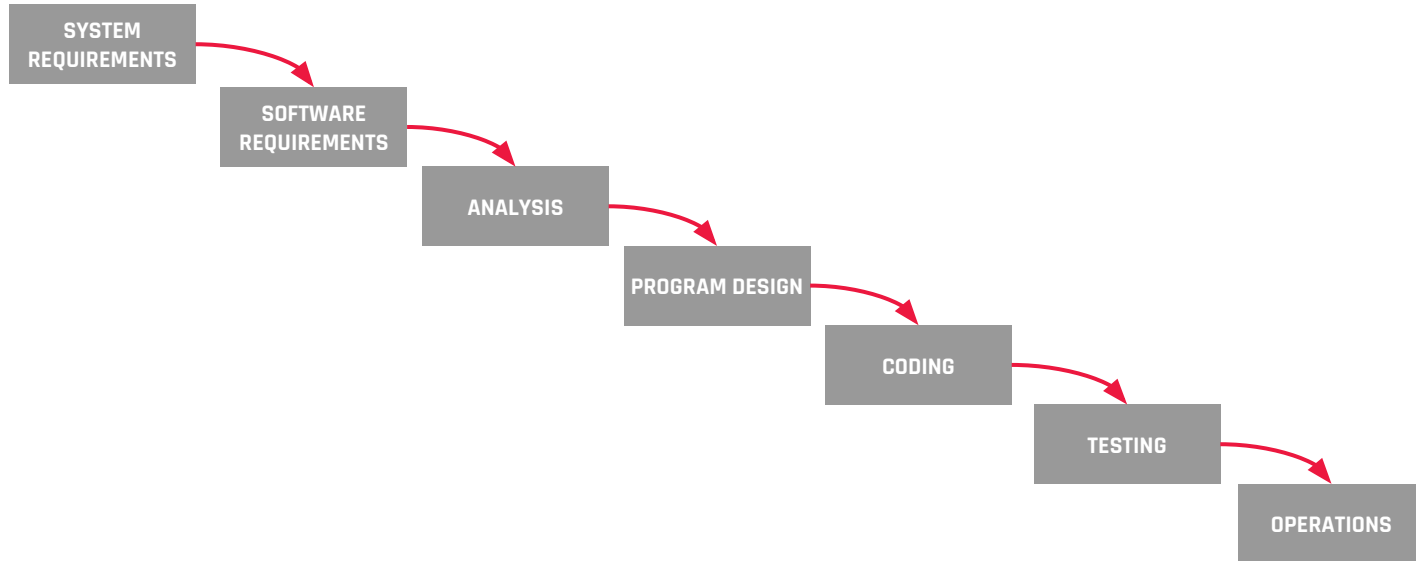
Em 1970, Winston Royce publicou um artigo sobre como **estruturar** projetos grandes e complexos, com base em suas experiências na NASA.

Esse artigo é considerado como a **definição** original do modelo sequencial de desenvolvimento de software, mas na realidade o que Winston queria fazer era apontar as deficiências que esse modelo apresentava e **propor melhorias**.



O modelo **cascata**

O modelo descrito por Winston é conhecido como **Modelo Cascata**, conforme demonstra o gráfico:



O que acontecia antes das metodologias ágeis?

Diz a lenda que, em 1985, um funcionário do DoD (Departamento de Defesa dos Estados Unidos) recebeu a tarefa de buscar um processo de desenvolvimento de software para **padronizar** os pedidos de projetos do departamento.

Essa pessoa encontrou o documento de Winston, leu as primeiras páginas, viu os gráficos, mas ignorou todos os avisos. 🤖

Foi a partir desse mesmo ano que o DoD estabeleceu essa metodologia como um **requisito** para qualquer empresa que quisesse escrever código para eles.

O que acontecia antes das metodologias ágeis?

Pode ser difícil imaginar a importância deste fato, mas ao estabelecer a Metodologia Cascata como padrão para qualquer projeto do DoD, outros **departamentos governamentais** começaram a fazer o mesmo, inclusive aqueles de outros países, sendo adotados posteriormente pelas empresas.



Hoje podemos dizer que o modelo publicado por Winston se tornou uma prática comum no desenvolvimento de software em todo o mundo.

O lado **positivo** da metodologia

Se pensarmos em suas origens, é lógico imaginar que ela funciona muito bem para ambientes rígidos onde há poucas mudanças, porque seu foco está na **prevenção** e mitigação dos **riscos** que afetam o **escopo** e o planejamento do projeto.

- Funcionam muito bem para processos onde não há incerteza;
- Facilita calcular prazos e custos do projeto.

O lado **negativo** da metodologia

Por outro lado, não se adapta bem a ambientes com muitas **incertezas** ou **mudanças**.

- Possui pouca capacidade de se adaptar às mudanças de requisitos;
- Todos os requisitos devem ser conhecidos desde o ponto zero do projeto;
- Como cada etapa é tão marcada, caso ocorra um atraso, todas as etapas seguintes, consecutivamente, sofrerão o atraso também.

Como surgiram as metodologias ágeis

A adoção das metodologias **sequenciais**

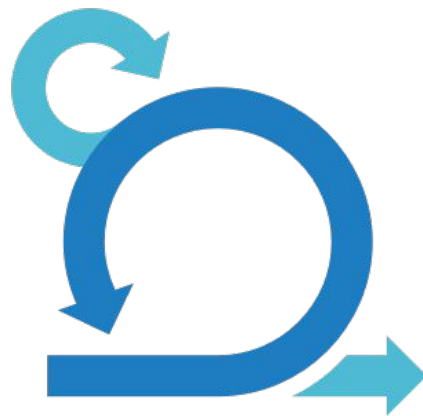
Em 1985, a **Metodologia de Cascata** foi estabelecida como padrão para qualquer pedido de desenvolvimento feito pelo DoD (Departamento de Defesa dos EUA).

Esse fato significou uma adoção **progressiva** da metodologia, a ponto de se tornar um padrão mundial no qual a maioria das pessoas envolvidas no desenvolvimento de software seria treinada como **líderes** de projetos, **proprietários** de produtos, **programadores, analistas**, etc.

A crise

Durante os anos seguintes, tornou-se cada vez mais evidente que metodologias sequenciais, como a Metodologia Cascata, **não** estavam **produzindo bons resultados**. Isso motivou a criação de novas metodologias de desenvolvimento, entre as quais podemos citar:

- Scrum;
- Extreme Programming (XP);
- Feature Driven Development (FDD);
- Crystal.



A crise

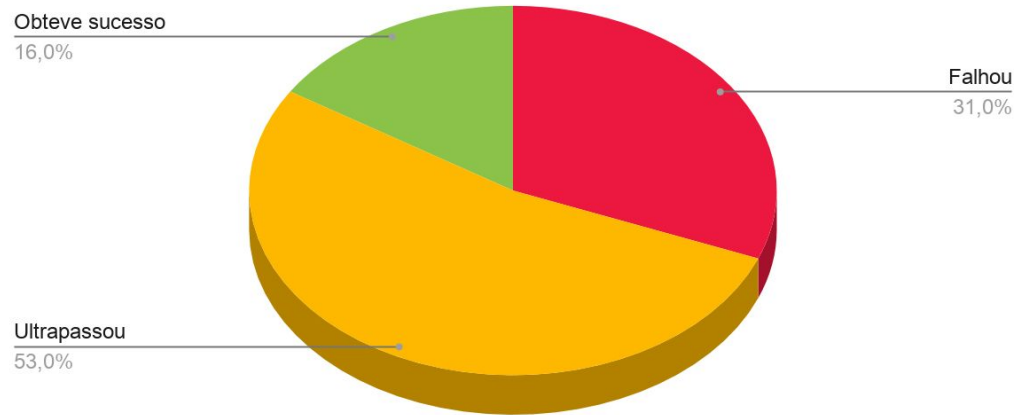
Em 1994, quase dez anos depois que a Metodologia de Cascata se tornou popular, foi publicado o **Relatório CHAOS** ("The CHAOS Report"). Um relatório elaborado com o resultado do estudo realizado em diferentes projetos de software desenvolvidos com a metodologia de cascata, em que apenas 16% foram considerados bem-sucedidos.



Isso quer dizer que, de cada 10 projetos iniciados, menos de 2 alcançaram seu fim ou cumpriram o planejamento original.

Relatório **CHAOS** de 1994

De acordo com o gráfico, os principais motivos pelos quais 84% dos projetos restantes foram **cancelados** ou **não cumpriram** as estimativas de tempo e custo iniciais, se devem às mudanças nos requisitos originais e ao fato de não terem envolvido os usuários do sistema durante a fase de desenvolvimento.



O Manifesto Ágil

Com isso, tornou-se claro que a abordagem com a qual os projetos de desenvolvimento de software eram **pensados** e **executados** precisava ser alterada.

Sendo assim, em 2001 os promotores das diferentes metodologias, que na época eram conhecidas como *soft*, *light* ou leve, concordaram na criação do que hoje é conhecido como **Manifesto Ágil**.

O Manifesto Ágil

O manifesto é essencialmente um **compromisso público** de encontrar melhores maneiras de desenvolver software e ajudar outros a implementá-lo. É composto por **4 valores**, **12 princípios** e o documento pode ser acessado pelo QR Code:



Os **4 valores** do Manifesto Ágil

Podemos resumir sua dinâmica de movimento lendo seus **4 valores**:

Indivíduos e interações
em processos e ferramentas

1

2

Software rodando
em extensa documentação

Respondendo a mudanças
sobre seguir um plano

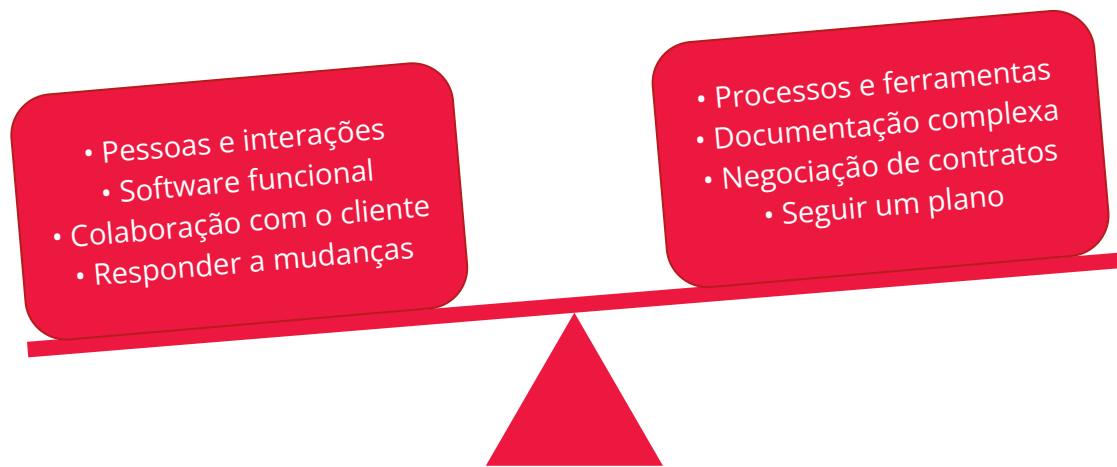
4

3

Colaboração com o cliente
na negociação contratual

Os 4 valores do Manifesto Ágil

Em outras palavras, embora valorizemos os elementos à direita, valorizamos ainda mais os da esquerda:



Características das metodologias ágeis

Como o próprio nome indica, o objetivo principal deste tipo de metodologia é promover agilidade em todos os aspectos do desenvolvimento de software. Vejamos então algumas de suas principais características:

Orientada para satisfação do cliente

O cliente está presente em todas as fases de desenvolvimento. Uma versão mínima do produto é entregue o mais rápido possível e, em seguida, essa primeira versão é aprimorada por meio de curtos ciclos de desenvolvimento.

Características das metodologias ágeis

Software em execução como medida principal de progresso

Cada ciclo tem como objetivo entregar uma versão adicional do projeto, o que não está finalizado não é entregue.

A mudança é bem-vinda

Os requisitos do projeto podem mudar por meio do feedback do cliente ou de situações externas, mesmo em estágios avançados.

Simplicidade

Buscar sempre maximizar o resultado e minimizar o esforço da equipe.

Características das metodologias ágeis

Equipes auto-organizadas

O foco de quem gerencia o projeto será criar um ambiente produtivo e com boa comunicação. A execução do projeto ficará a cargo da equipe de desenvolvimento e da forma que julgar adequada.

Melhoria contínua

Através da reflexão regular sobre o trabalho realizado, os processos implementados e a experiência da equipe.

Características das metodologias **ágeis**

Desta forma, temos um processo de desenvolvimento que evolui naturalmente para a sua **melhor versão** à medida que as **necessidades** do **cliente** são mais conhecidas e a equipe ganha experiência como um time.



Assim, se promove o desenvolvimento sustentável e um ritmo constante de trabalho pode ser mantido indefinidamente.

Cascata vs Agile

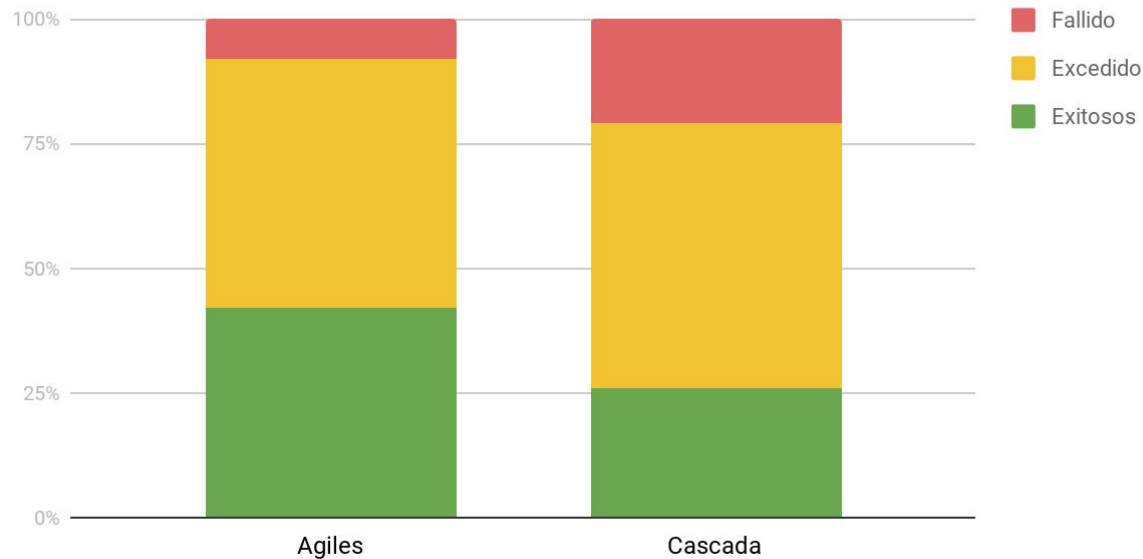
Em geral, considera-se que um projeto desenvolvido com metodologias ágeis tenha **duas vezes mais chances de sucesso** do que um projeto com metodologia em cascata.

As versões recentes do Relatório CHAOS mostram uma clara diferença entre as duas metodologias e atualmente as metodologias ágeis estão se tornando cada vez mais comuns, tanto para projetos de desenvolvimento de software quanto para outras indústrias.

Cascata vs Agile

Metodologías de Cascada vs Ágiles

Resultados del CHAOS Report de 2019



Cascata vs Agile

PROJECT SUCCESS RATES AGILE VS WATERFALL



WWW.VITALITYCHICAGO.COM

Source: Standish Group Chaos Studies 2013-2017

“ Ciclos curtos de desenvolvimento e vontade de mudar geram uma vantagem competitiva em relação a projetos rígidos e um produto mais adaptado às necessidades do cliente e do mercado. ”



SCRUM

DigitalHouse >
Coding School

Índice

1. O que é SCRUM?
2. Funções da equipe
3. Documentos
4. Cerimônias

1 | O que é SCRUM?

O que é **SCRUM**?

Scrum é uma das metodologias de **desenvolvimento ágil** mais utilizadas no mundo. Sua popularidade se deve ao fato de ser fácil de entender e implementar, pelo menos em sua versão mais simples, pois os resultados são evidentes desde os **estágios iniciais** do projeto.

Em todo caso, não precisamos nos deixar levar por sua relativa simplicidade, para dominar a metodologia e para que ela produza os melhores resultados, é preciso tempo e muita prática.

O que é **SCRUM**?

Scrum é apresentado como um **framework**, não é um processo em si. Em outras palavras, o Scrum não diz à equipe como fazer as coisas, ao invés disso, ele ajuda a **priorizar** o que precisa ser feito e deixa a responsabilidade de determinar como fazer.

Na mesma linha de pensamento, Scrum não obriga a **implementação** da metodologia da forma como é apresentada, mas sim a oferece como uma **caixa de ferramentas** e convida a equipe a utilizar os elementos que melhor se adequa à sua forma de trabalhar, abrindo mão do que não atende ao propósito.

“ Scrum foca em fomentar o trabalho em equipe, a responsabilidade e a visibilidade do trabalho (passado, presente e futuro), tudo isso gerando um progresso iterativo em direção a um objetivo bem definido. ”



2 | Funções da equipe

Product owner

Representa a voz do **cliente** e é aquele que toma as decisões finais do produto.

Garante que a equipe trabalhe corretamente do ponto de vista de **negócios**.

Escreve histórias de usuários, prioriza-as e coloca-as no Product Backlog (veremos esses significados mais adiante).

Scrum master

É um **facilitador** cujo foco principal é garantir uma troca eficiente de informações. Ele é responsável por controlar o **tempo**, as **conversas** e o **processo** em geral.

Ajuda a remover **obstáculos** e atua como um buffer entre equipe e distrações. Verifica se cada membro da equipe usa suas habilidades únicas para o sucesso da equipe.



É importante deixar claro que o **Scrum master** não é o líder da equipe.

Dev Team

Também chamada de “equipe de desenvolvimento”, é composta por um grupo de 3 a 9 pessoas que devem possuir todas as **competências técnicas** necessárias para a entrega do produto ou serviço que está sendo desenvolvido.

Embora sejam orientados pelo Scrum Master e recebam suas tarefas do Product Owner, não haverá ninguém para lidar com elas diretamente. Por dentro, as equipes de trabalho são horizontais e todas as decisões são tomadas por consenso. As equipes Scrum são **auto-organizadas**, versáteis e responsáveis o suficiente para completar todas as tarefas exigidas delas.

Stakeholders

Os stakeholders (ou acionistas) incluem todas as pessoas que **usarão**, ou de alguma forma **serão afetadas**, por nosso produto ou serviço.

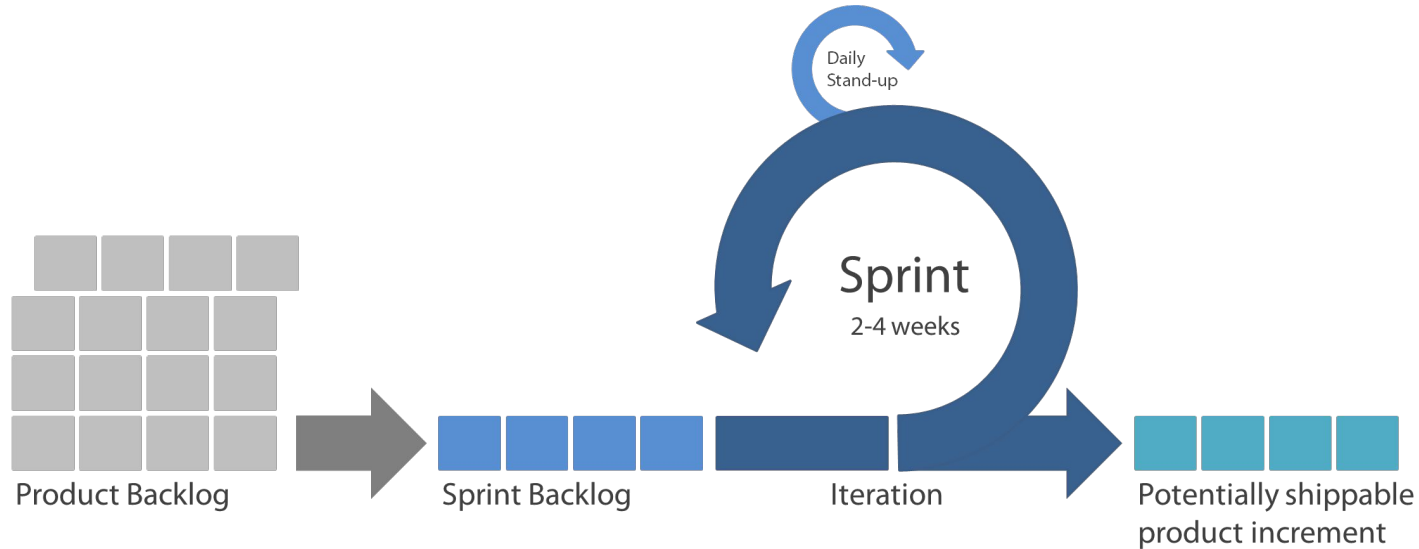
Embora não façam parte da equipe Scrum, é bom nomeá-los, pois durante o processo de desenvolvimento estaremos direta ou indiretamente em contato com eles.

3

Documentos

Documentos

Documentos são usados para definir e gerenciar o **fluxo de trabalho** recebido pela equipe.



Product backlog

É o conjunto de todos os **requisitos** do projeto.

Ele contém descrições genéricas dos recursos desejáveis, priorizados de acordo com seu retorno sobre o investimento (ROI). O projeto que causa o **maior impacto positivo** para os clientes será desenvolvido primeiro.
Representa a totalidade do que será construído.

Normalmente, essa lista é gerenciada pelo **Product Owner**.

Sprint

Sprint é o **período** em que o trabalho em si é realizado.

Sua duração é sempre a mesma e é definida pela equipe. Geralmente, entre **2 e 4 semanas**.

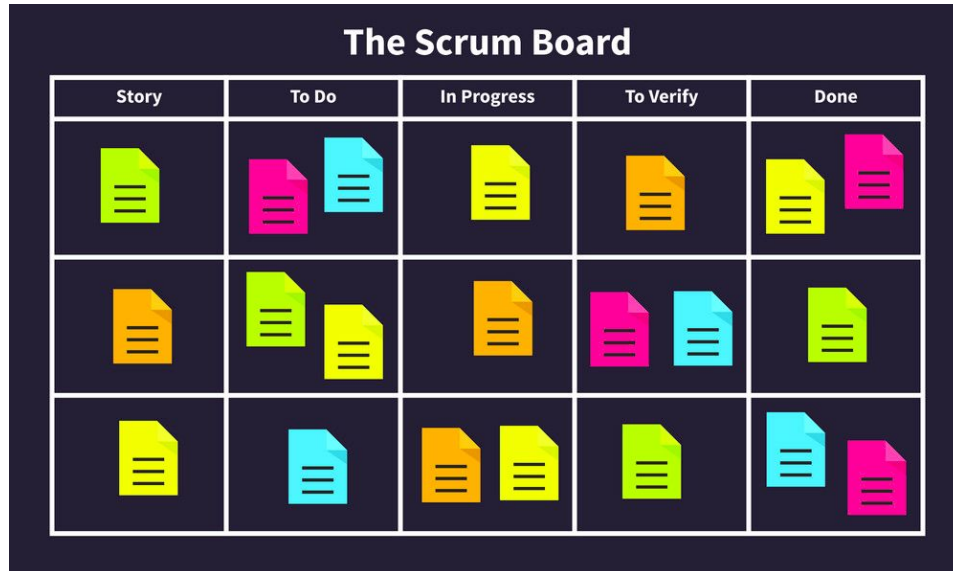
Sprint backlog

É o subconjunto de requisitos que serão **desenvolvidos** durante o sprint atual.

Essa lista é gerenciada pela equipe de desenvolvimento. Isso significa que a equipe determinará quantas **tarefas** serão concluídas durante o sprint.

Scrum board

Existem várias versões e é comum o uso de ferramentas digitais como **Trello**, **Asana**, **Monday**, etc.



Scrum board

Normalmente, vemos as seguintes colunas:

- **História (Story)** : Histórias de usuários que deram origem às tarefas;
- **Pendentes (To Do)**: as tarefas a serem realizadas no sprint;
- **Em progresso (Doing)**: as tarefas em andamento neste sprint;
- **A revisar (To Verify)**: tarefas prontas que requerem verificação;
- **Concluídas (Done)**: tarefas prontas e verificadas (critérios de aceitação aprovados).

História de usuários

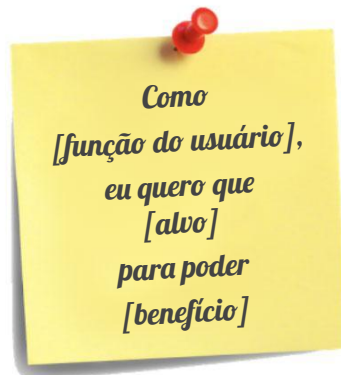
Nos permitem descrever uma **funcionalidade** ou comportamento visto pelos olhos do usuário (**empatia**).

As histórias de usuário devem ter **critérios de aceitação**. Ou seja, condições que devem ser atendidas para encerrar a tarefa que descrevem.

Eles geralmente têm o seguinte formato:

Como [função do usuário], eu quero que [alvo] para poder [benefício].

Várias tarefas para a equipe de desenvolvimento podem surgir de uma história de usuário.



4 | Cerimônias

“ Visam manter a
comunicação da equipe e
garantir visibilidade das
tarefas e obstáculos que
podem ser encontrados. ”



Planning

Durante o **planejamento**, se define o trabalho a ser feito durante o sprint.

Essa cerimônia é feita com toda a equipe, onde é acordado o trabalho a ser feito durante o sprint.

Ele identifica e comunica quanto **esforço** provavelmente será **investido** nas diferentes **tarefas** propostas.

Normalmente, um máximo de **8 horas** é calculado para um sprint de 1 mês.

Daily

É uma reunião em que cada membro da equipe fala por, no máximo, **3 minutos**. Tem o objetivo de responder a três perguntas simples:

- O que eu fiz ontem?
- Tive algum impedimento para alcançar meus objetivos?
- O que vou fazer hoje?

As reuniões costumam ser feitas em **pé** para incentivar a **velocidade**.

Sprint Review

O objetivo desta reunião é **revisar** o trabalho que foi concluído e o que não foi. Em seguida, apresentar o trabalho concluído às partes interessadas.

Trabalho incompleto (que não agrega valor) não pode ser demonstrado.

A reunião não deve durar mais de **4 horas**, para um sprint de 1 mês.

Sprint Retro

Diferente das demais reuniões, que focam no trabalho a ser feito ou realizado, esta tem como foco a **melhoria contínua** do processo e a comunicação da equipe.

Existem muitas dinâmicas para realizar nesta reunião: estrela do mar, linha do tempo, etc.

A duração desta reunião é de **4 horas** fixas para um sprint de 1 mês.

Wireframes

O que vimos no Playground

- O que são wireframes
- Opções de ferramentas para a criação
- Apresentação Sprint 0



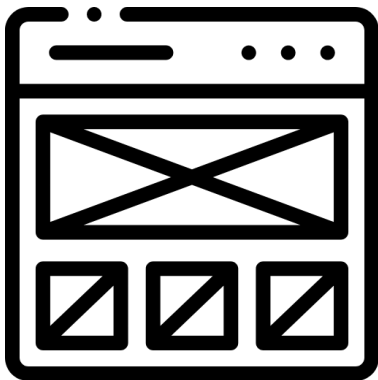
O que vamos ver hoje

- Apresentação ferramentas de apoio
- Criação de wireframe
- Apresentação Sprint 0



O QUE SÃO **WIREFRAMES**?

Qual destas imagens são consideradas wireframes?

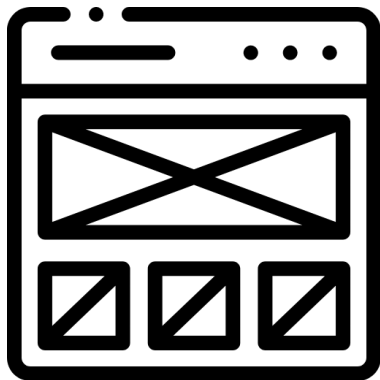


A)

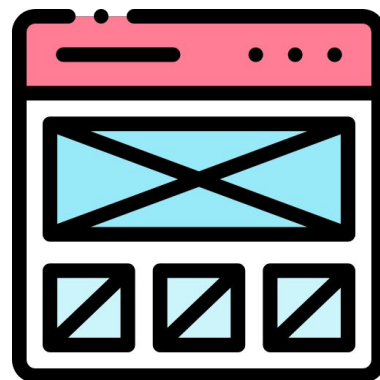


B)

E dentre estas imagens?



A)



B)

Estão lembrados das ferramentas sugeridas no material?



Como fazer um wireframe

“

O objetivo do **wireframe** é **mostrar** onde os **elementos de uma página** irão **aparecer** e como o usuário irá interagir com ela. ”



Índice

1. Elementos de um wireframe
2. Os componentes
3. Construindo uma página

1

Elementos de wireframe

As cores

Os **wireframes**, especialmente os de **baixa fidelidade**, geralmente trabalham com uma paleta de cores muito limitada.

Normalmente encontramos transparente, branco, escala de cinza e preto. Opcionalmente, podemos trabalhar com uma cor de destaque.

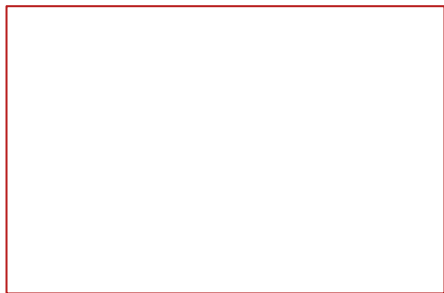


opcional

Blocos e separações

Servem para **delimitar espaços** e **agrupar elementos**.

Eles podem ter bordas, fundos coloridos e arredondamentos para diferenciá-los.



Títulos

Estes são textos destacados, como o **título do site, o título de uma seção, o título de um produto ou artigo, etc.**

Normalmente, eles trazem um texto real se forem elementos importantes, para o resto, ele pode ser inventado.

Título principal (h1)

Título secundário (h2)

Título 3 (h3)

Título 4 (h4)

Título 5 (h5)

Título 6 (h6)

Parágrafos e listas

São blocos de texto normais.

Eles podem ter um **texto real** ou de **preenchimento**.

Para o texto de preenchimento, podemos usar [este site](#), que é um gerador de textos 'aleatórios'.

Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Curabitur sit
amet consequat enim.

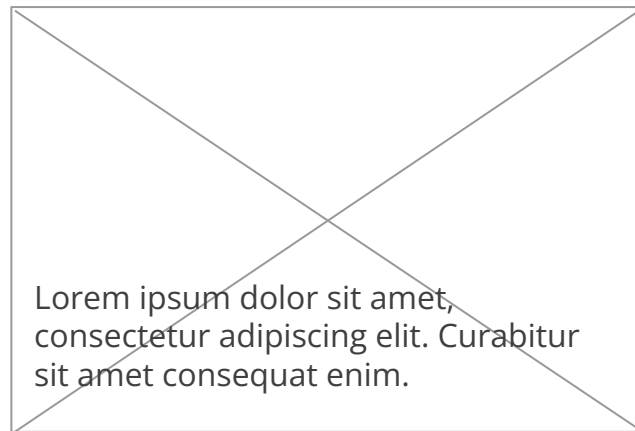
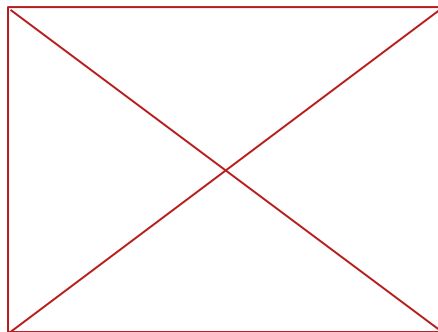
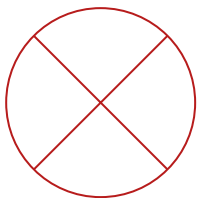
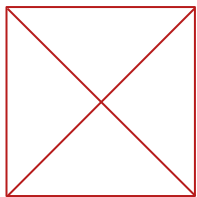
Duis urna turpis, fringilla ut ornare ac,
mollis in leo. Suspendisse vestibulum
dictum vehicula.

- Lorem ipsum dolor sit amet.
- Consectetur adipiscing elit.
- Curabitur sit amet consequat.

1. Duis urna turpis,
2. Fringilla ut ornare ac,
3. Mollis in leo.

Imagens

Elas são representadas por um **quadro com uma cruz atravessada**.
Podem ser do tamanho que precisarmos e também podem ser redondas.
Podem ser imagens de fundo e ter outros elementos em cima.



Elementos do formulário

Normalmente, nesta fase, tentaremos usar os elementos mais básicos dos formulários, mas podemos criar aquele que necessitamos.

Etiqueta:

 ▼☐

Checkbox

link

☐

Radio

Comentários

Também podemos acrescentar comentários para descrever um elemento ou o que deve acontecer quando o usuário interage com ele.



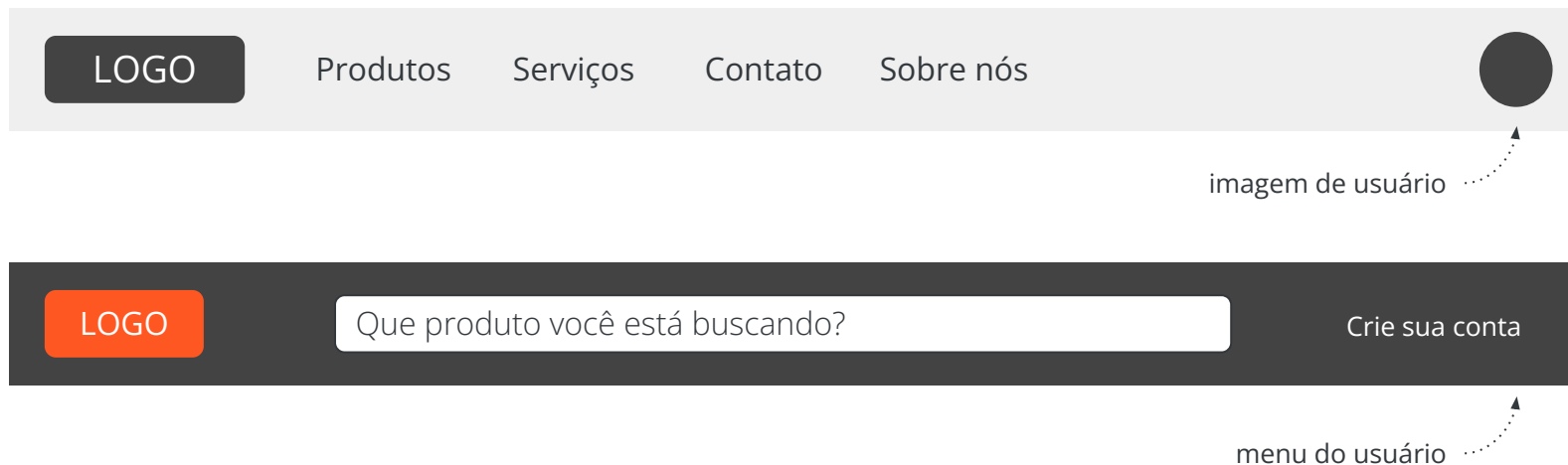
2 | Os componentes

“ Utilizando os **elementos básicos** que nos propõe um wireframe, podemos **construir componentes** mais complexos. ”



O header

Normalmente teremos um **logotipo**, um **menu de navegação** e um **sub-menu**. O logotipo pode ser uma imagem ou um bloco com a palavra "logo".



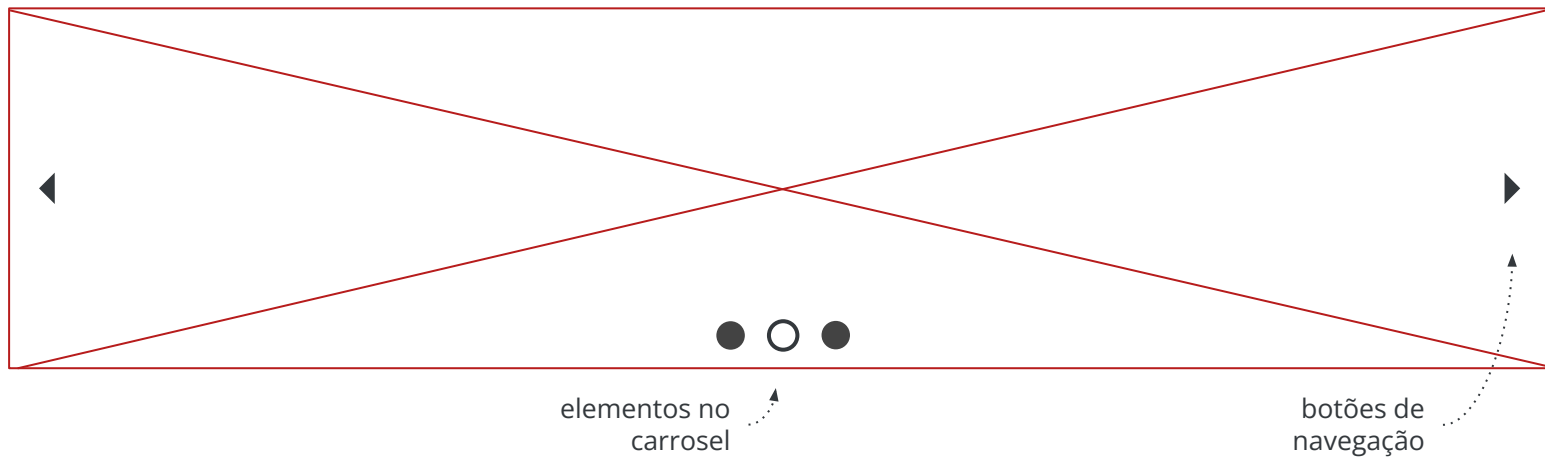
0 footer

Geralmente teremos várias colunas de links.
Também podemos ter logotipos de redes sociais.



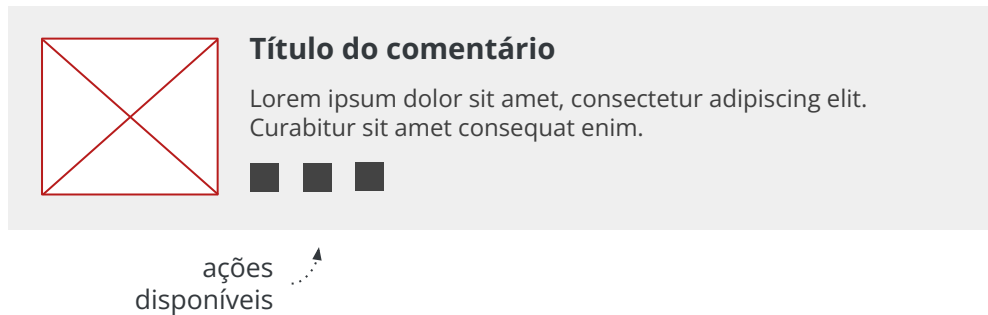
0 carrossel

Imagens que vão passando automaticamente.



Blocos de conteúdo

Um dos elementos mais comuns de qualquer site. Podem ter imagens, títulos, textos, botões e tudo mais que precisarmos.



Formulários


Outro dos elementos obrigatórios de qualquer site.

Crie sua conta!

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Email

Nome



3

Construindo uma página

Uma página completa

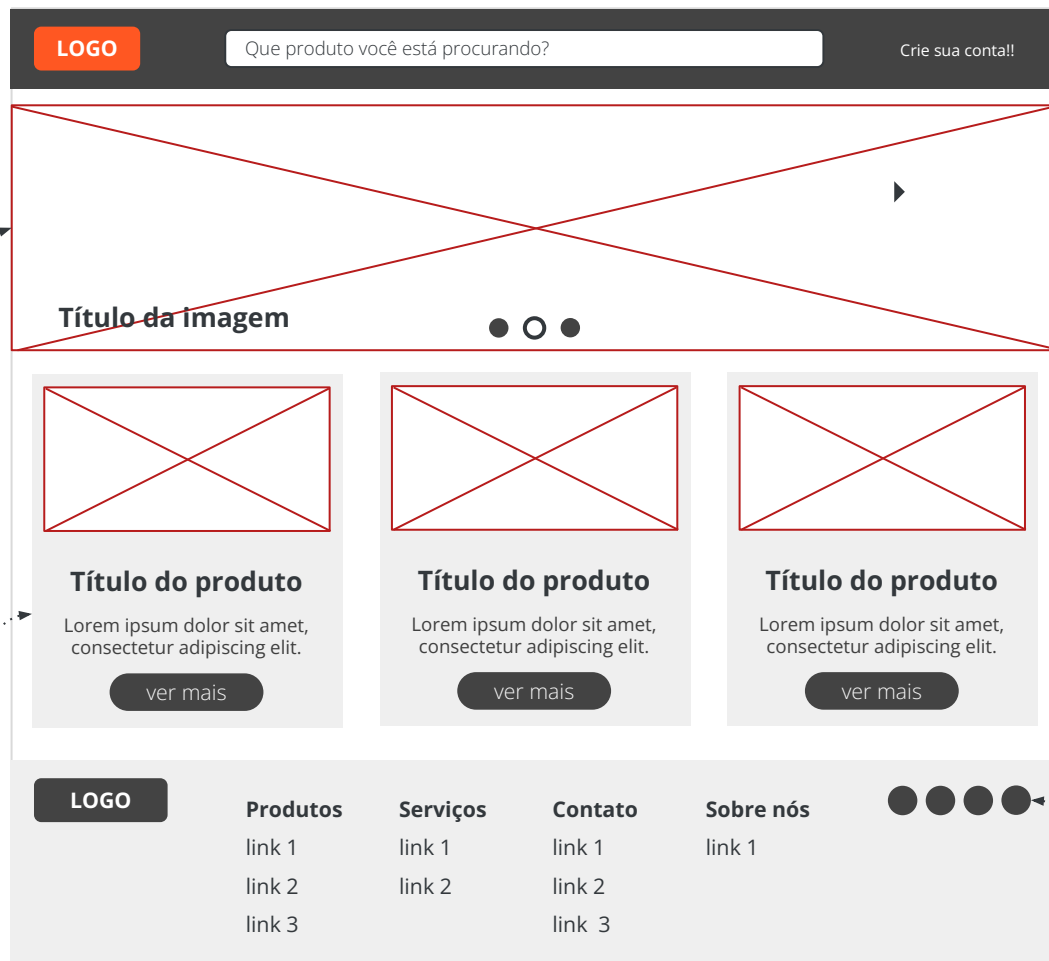
Para montar uma página completa, basta juntar todos os componentes que vimos anteriormente.



Página principal

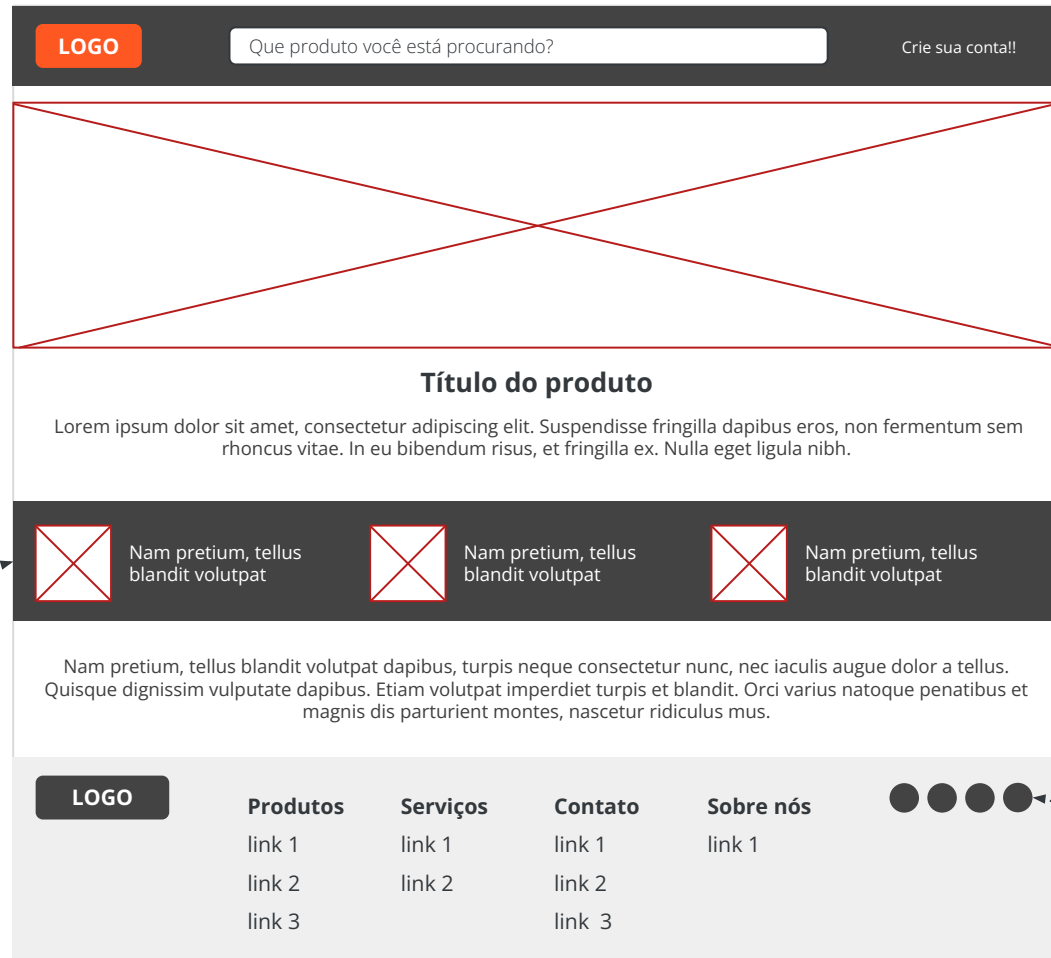
carrossel com produtos que estão em oferta

produtos de destaque do mês



Detalhe produto

características principais do produto



redes sociais

Formulário de registro

LOGO

Que produto você está procurando?

Crie sua conta!!

Crie sua conta

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Email

yo@nofui.com

Nome

Barto

criar conta

LOGO

Produtos
link 1
link 2
link 3

Serviços
link 1
link 2

Contato
link 1
link 2
link 3

Sobre nós
link 1

redes
sociais

Ferramentas para fazer wireframes



**Apresentamos várias ferramentas
que podemos utilizar para montar
nossos **wireframes**.**

Wireframe.cc



É uma ferramenta online específica para wireframing. É muito simples de usar. Você precisa se registrar para poder salvar em formato de imagem.

Link: <https://wireframe.cc>

Formato: online

Custo: grátis/assinatura

Dificuldade: ★★★★★

diagrams.net



É uma ferramenta online e de mesa que permite a montagem de todos os tipos de diagramas, incluindo wireframes.

Fornece exemplos pré-montados de wireframes para a web.

Link: <https://diagrams.net>

Formato: online e desktop

Custo: grátis

Dificuldade: ★★☆☆☆

Miro



É uma ferramenta on-line que permite montar todos os tipos de diagramas.

Frequentemente usado para trabalhar de maneira colaborativa.

Link: <https://miro.com>

Formato: online

Custo: grátis/assinatura

Dificuldade: ★★★★★

Marvel



É uma ferramenta on-line que permite montar protótipos e também pode ser usada para elaborar wireframes.

É um pouco mais complexa, mas nos permite construir um protótipo interativo.

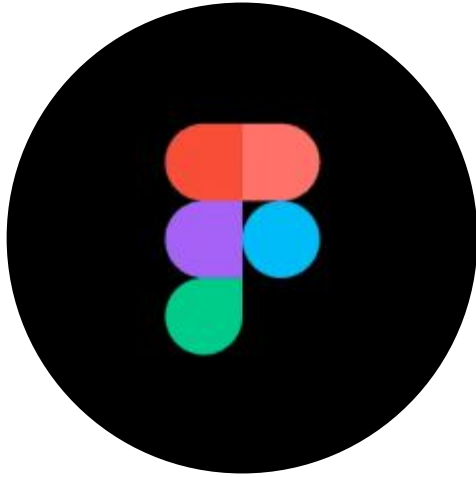
Link: <https://marvelapp.com>

Formato: online

Custo: grátis/assinante

Dificuldade: ★★☆☆☆

Figma



É uma ferramenta online que permite montar protótipos interativos e também pode ser usada para construir wireframes. Frequentemente usada para trabalhar de maneira colaborativa.

Link: <https://www.figma.com>

Link: [Kit de wireframing para Figma](#)

Formato: online

Custo: grátis /assinatura

Dificuldade: ★★★★★

Sketch



É uma das melhores ferramentas para design de interfaces.

É usada por profissionais de design para criar todo tipo de interfaces.

Link: <https://www.sketch.com>

Formato: desktop (apenas para Mac)

Custo: licença

Dificuldade: ★★★★★

Adobe Xd



É outra das melhores ferramentas para design de interfaces.

É usada por profissionais de design para criar todo tipo de interfaces.

Link: <https://adobe.com/br/products/xd.html>

Formato: desktop (Mac e Windows)

Custo: amostra grátis/assinatura

Dificuldade: ★★★★★

