UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC004 - Algoritmos e Estruturas de Dados II

Construção, depuração e revisão de código

Renato Martins

Email: renato.martins@dcc.ufmg.br

https://www.dcc.ufmg.br/~renato.martins/courses/DCC004

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Processo de compilação
Quando fazemos de forma manual

- Até o momento estamos compilando todo o código de uma vez
- Porém o mesmo é composto de módulos que são independentes
- Trabalhamos com tudo em uma pasta

```
$ g++ arq1.cpp arq2.cpp -o main
ou
$ g++ *.cpp -o main
```

Formas de organizar o código
Melhor organização final

Agrupar o código em pastas lógicas

```
. raiz do projeto
|---- Makefile
|---- build/                          [diretório]
    |----                            [objetos compilados]
|---- include/                        [diretório]
    |---- modulo1/                    [diretório]
        |---- modulo1_arquivo1.h      [cabeçalho]
        |---- modulo1_arquivo2.h      [cabeçalho]
    |---- modulo2/                    [diretório]
        |---- modulo2_arquivo1.h      [cabeçalho]
        |---- modulo2_arquivo2.h      [cabeçalho]
|---- src/                            [diretório]
    |---- modulo1/                    [diretório]
        |---- modulo1_arquivo1.cpp    [código]
        |---- modulo1_arquivo2.cpp    [código]
    |---- modulo2/                    [diretório]
        |---- modulo2_arquivo1.cpp    [código]
        |---- modulo2_arquivo2.cpp    [código]
```

Módulos + Namespaces
Úteis quando temos nomes repetidos

- Imagine uma biblioteca em C++ para diferentes tipos de jogos de carta
- Em todos os jogos temos:
  - Mãos
  - Jogadores
  - Baralhos

- Porém as cartas são bem diferentes

Organizando o Código
Note que temos definições repetidas

```
. raiz do projeto
|---- Makefile
|---- build/                    [diretório]
    |----                       [objetos compilados]
|---- include/                  [diretório]
    |---- magic/                [diretório]
        |---- carta.h           [cabeçalho]
    |---- uno/                  [diretório]
        |---- carta.h           [cabeçalho]
|---- src/                      [diretório]
    |---- magic/                [diretório]
        |---- carta.cpp         [código]
    |---- uno/                  [diretório]
        |---- carta.cpp         [código]
```

Namespaces
Uma ajuda com definições repetidas

- Situações como estas são comuns em um software grande
- Namespaces ajudam a resolver o problema
- Podemos ter nomes repetidos em namespaces diferentes

Dois cartas.h, cartas.cpp e Classes Carta

include/uno/carta.h

```
#ifndef PDS2_CARTAUNO_H
#define PDS2_CARTAUNO_H
namespace uno {
  enum Cor {AZUL, VERDE, AMARELO, VERMELHO};

  class Carta {
  private:
    Cor _cor;
    int _numero;
  public:
    Carta(Cor cor, int numero);
    int get_numero() const;
    Cor get_cor() const;
  };
}
#endif
```

Uso

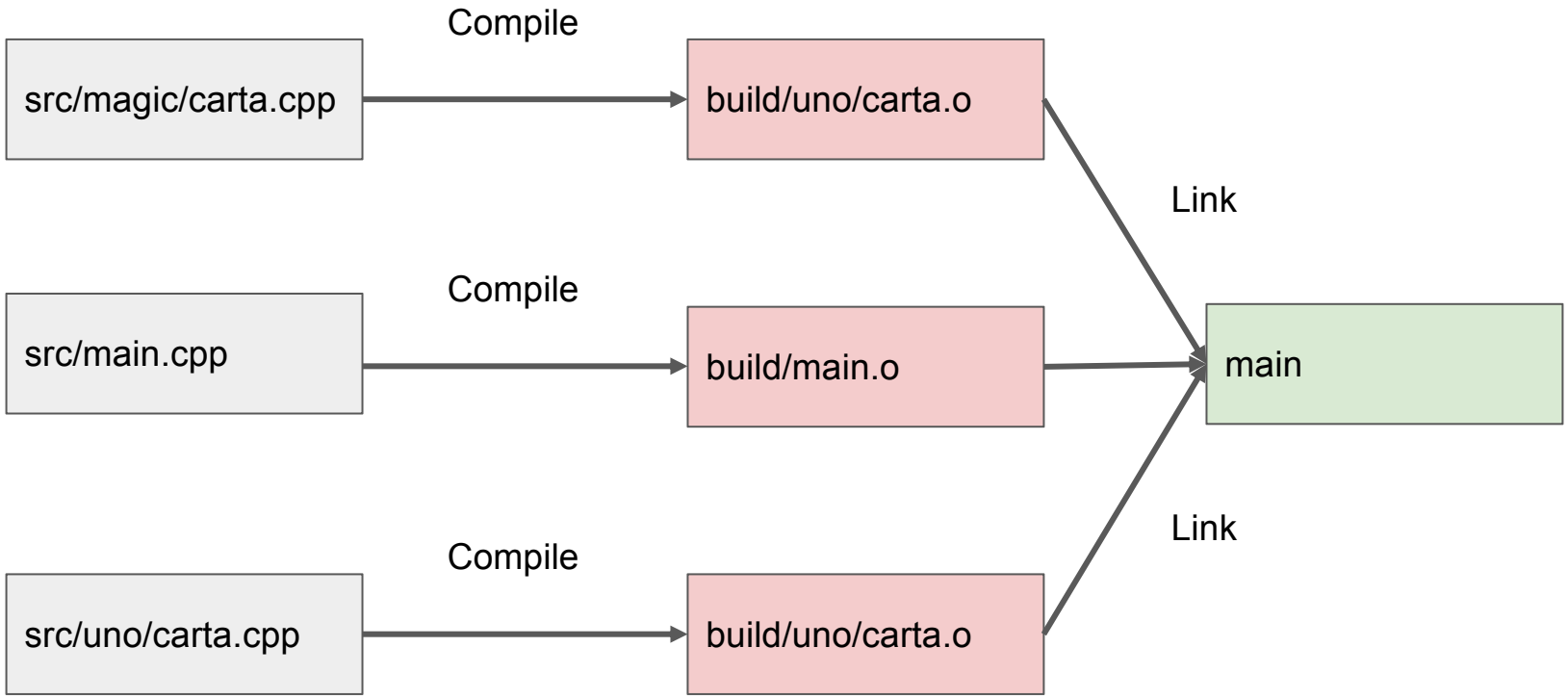
- Diferenciamos os arquivos com o namespace. Estilo fazemos com o std

```
#include <string>

#include "magic/carta.h"
#include "uno/carta.h"

int main() {
  uno::Carta carta_uno(uno::Cor::VERMELHO, 12);
  magic::Carta carta_magic("Monster", 72.1);
}
```

Compilando Programas Maiores

Módulos podem ser compilados de forma independente

- Compilação ocorre em várias etapas
- Não observamos ao compilar tudo de uma vez



Automatizando Makefiles

- Arquivos de compilação automática
- Indica as dependências entre os módulos
- Serve para automatizar desenvolvimento
- Conjunto de regras e dependências

```
target1 target2 ... : dependencia1 dependencia2 ...
    <TAB> comando1
    <TAB> comando2
    ...
```

```
helloworld: helloworld.cpp
    g++ -o helloworld helloworld.cpp
```

Makefile

```
CC := g++
SRCDIR := src
BUILDDIR := build
TARGET := main
CFLAGS := -g -Wall -O3 -std=c++11 -I include/

all: main

magic:
	@mkdir build/magic/
	$(CC) $(CFLAGS) -c src/magic/carta.cpp -o build/magic/carta.o

uno:
	@mkdir build/uno/
	$(CC) $(CFLAGS) -c src/uno/carta.cpp -o build/uno/carta.o

main: magic uno
	$(CC) $(CFLAGS) build/magic/carta.o build/uno/carta.o src/main.cpp -o main

clean:
	$(RM) -r $(BUILDDIR)/* $(TARGET)
```

Makefile

- Mais um código para manter
- É possível fazer Makefiles mais genéricos
  - Funcionam em projetos que seguem a mesma organização

Lembrando
Tipos de erros

- Uma forma de ver:
  - Erros de sintaxe
  - Erros de semântica
  - Erros de lógica
- Outra forma:
  - Erros em tempo de compilação
  - Erros em tempo de execução

Erros de Sintaxe
Exemplos

- Quase sempre são erros de compilação

```
#include <iostream>

int main() {
  st::cout << "oi";
  return 0;
}
```

Erros de Semântica
Exemplos

- Podem ou não gerar um erro/warning

```
#include <iostream>

int *f() {
  int *rv[20];
  return rv;
}
```

Erros de Lógica

Não temos mais ajuda do compilador

```
#include <iostream>
#include <cmath>

int fatorial(int num) {
  int fat = 0;
  for (int i = 1; i <= num; i++)
    fat = fat * i;
  return fat;
}

double series(double x, int n) {
  double valor = 0.0;
  double xpow = 1;
  for (int k = 0; k <= n; k++) {
    valor += xpow / fatorial(k);
    xpow = xpow * x;
  }
  return valor;
}
```

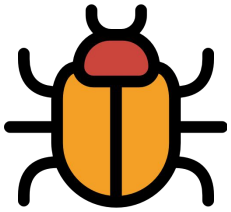
Erros de Lógica

Não temos mais ajuda do compilador

```
#include <iostream>
#include <cmath>

int fatorial(int num) {
  int fat = 0;
  for (int i = 1; i <= num; i++)
    fat = fat * i;
  return fat;
}

double series(double x, int n) {
  double valor = 0.0;
  double xpow = 1;
  for (int k = 0; k <= n; k++) {
    valor += xpow / fatorial(k);
    xpow = xpow * x;
  }
  return valor;
}
```

Fatorial sempre zero

Depuração
Motivação



- Encontrar erros em códigos é uma arte
- Com o tempo:
  - aprendemos a mapear comportamento anômalo para erros comuns
  - o depurador nos ajuda
- Vamos usar o GDB para o erro acima

GNU Project Debugger

- GDB é o depurador "padrão" de C/C++
- Pode ser utilizado pela linha de comando
- Ou com uma GUI
  - http://gdbgui.com
- Ou em uma IDE
  - CodeBlocks e Visual Studio
- Até online

  https://www.onlinegdb.com/

Passos para Depurar o Código

- Reproduzir o problema
- Achar o local do erro
- Verificar efeitos colaterais
- Testar o código novamente
- Resolver

Dicas Gerais

- Entenda as mensagens de erro
- Pense antes de escrever
- Procure por problemas comuns
- Dividir para conquistar
- Mostre o valor de variáveis importantes
- Utilize um depurador
- Concentre-se em mudanças recentes

Dicas Ruins

- Encontrar defeitos adivinhando (na sorte)
- Fazer alterações aleatórias até funcionar
- Não fazer um backup do original e não manter um histórico das alterações feitas
- Corrigir o erro com a solução mais óbvia sem entender a razão do problema
- Se o erro sumiu, tudo ok!

Erros de Memória

- Memory Leaks
  - Código sem delete
- Acessos inválidos
  - Acesso para null
  - Acesso para memória desalocada
- Arquivos abertos
  - Arquivos sem close

Erros de Memória

- Para erros de memória podemos fazer uso de ferramentas como
- Valgrind
  - Comum em ambientes Unix
- DrMemory
  - Disponível para Windows/Linux/Mac
- Dicas de como instalar:
  - https://github.com/flaviovdf/programacao-2/tree/master/valgriddrmem

Revisão de Código

- Analisar o código de uma outra pessoa
- Geralmente feito antes de código ir para o repositório
  - Push no Github
- Série de ferramentas ajudam nesta tarefa

Revisão de Código

Quem

- Desenvolvedor do código e o responsável pela revisão (desenvolvedor mais experiente), às vezes juntos pessoalmente, às vezes separados

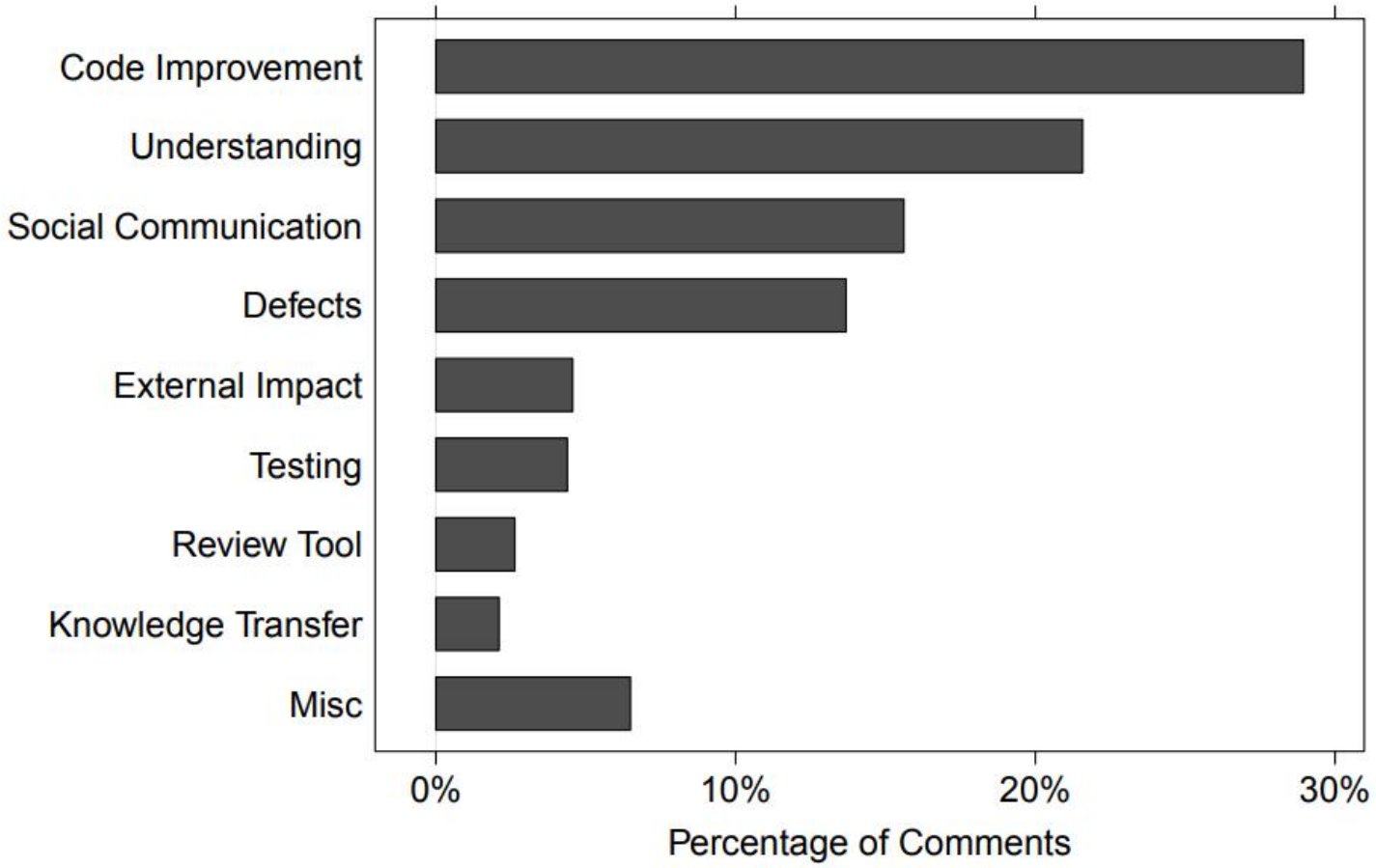
Como

- Revisor dá sugestões de melhoria em um nível lógico e/ou estrutural, de acordo com conjunto previamente acordado de padrões de qualidade
- Correções são feitas até uma eventual aprovação do

Quando

- Após o autor de código finalizar uma alteração do sistema (não muito grande/pequena), que está pronta para ser incorporada ao restante

Code Review



Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. ICSE.

Modern Code Review @ Google

https://ai.google/research/pubs/pub47025

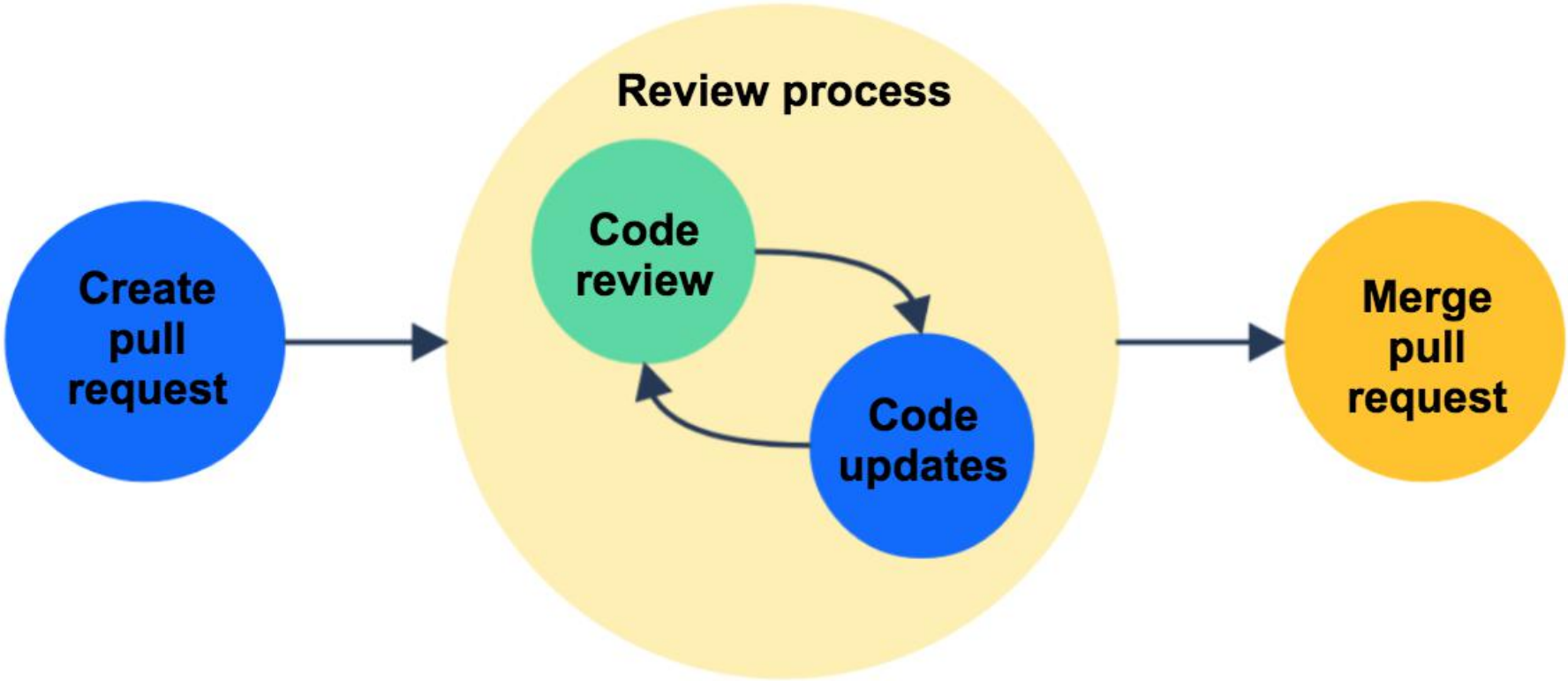
"All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian [Rietveld] to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

– Amanda Camp, Software Engineer, Google

Tipos de Revisão

- **Email**
  - Olá, olhe meu código.
- **Ferramentas**
  - Gerrit
  - Rietveld
- **Ciclo de pull requests**
  - Github

Revisão de Código
No Github



Controle de Versão

- Voce já:
  - Fez uma mudança no código, e percebeu que estava errada e quis voltar?
  - Manter múltiplas versões do mesmo código?
  - Quis ver a diferença entre duas (ou mais) versões do seu código?
  - Quis ver se uma mudança em particular quebrou ou consertou seu código?

Controle de Versão

- Mantém um registro das modificações
- Permite:
  - Desenvolvimento colaborativo
  - Saber quem fez as mudanças e quando
  - Reverter mudanças, voltar a um estado anterior
- Soluções livres
- Git, Mercurial, SVN

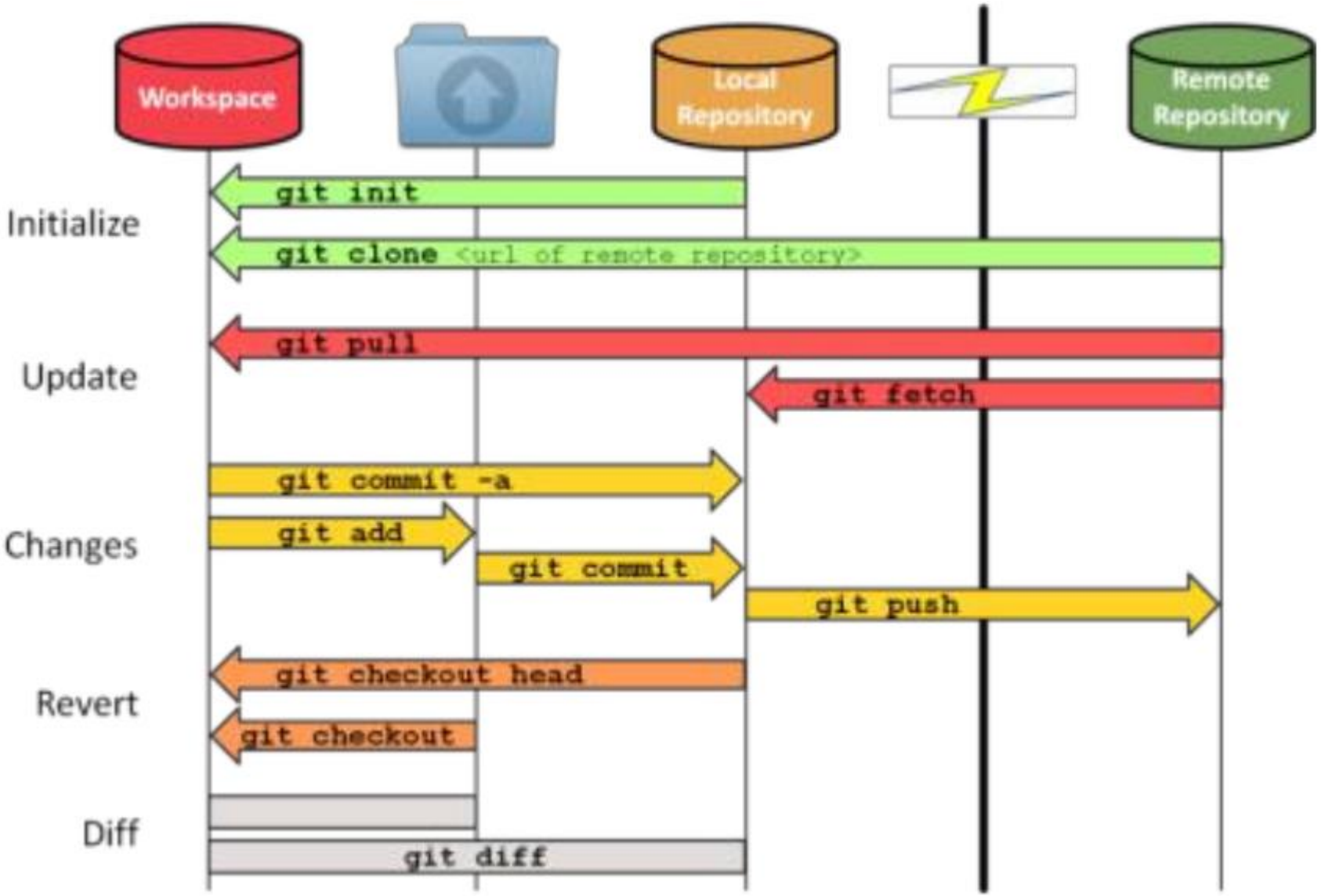
Controle de Versão
Git

- Sistema de controle de versões distribuído
- Desenvolvimento do kernel do Linux
- Github:
  - Serviço para armazenar repositórios
  - Se você é estudante não precisa pagar
  - https://education.github.com/pack

Controle de Versão: Git

- Repositório
  - Utilizado para organizar um único projeto
- Branch
  - Maneira de trabalhar em diferentes versões de um repositório de uma só vez
- Commit
  - Registro de quais arquivos você alterou desde a última vez que você fez um commit

Controle de Versão: Git



Tarefa: Criem uma conta pessoal no Github