

Programação e Desenvolvimento de Software 2

Versionamento e Revisão de Código

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Versionamento de Código

Gerência de Configuração de Software

- ☐ Durante o processo de desenvolvimento de software, nós queremos saber:
 - ☐ O que mudou?
 - ☐ Quando mudou?
 - ☐ Por que mudou?
 - ☐ Quem fez essa mudança?
 - ☐ Podemos reproduzir essa mudança?
 - ☐ Podemos recuperar o estado anterior à mudança?

Gerência de Configuração de Software

- ☐ Identificação
- ☐ Documentação
- ☐ Controle
- ☐ Manutenção
- ☐ Auditoria
- ☐ Artefatos:
 - ☐ Código Fonte
 - ☐ Documentação do Sistema
 - ☐ Manual do Usuário

Gerência de Configuração

☐ Problema Exemplo:

- ☐ Você precisa editar um código que está no seu Dropbox
- ☐ Você faz o download do arquivo
- ☐ Faz as alterações necessárias
- ☐ Salva novamente o arquivo no dropbox

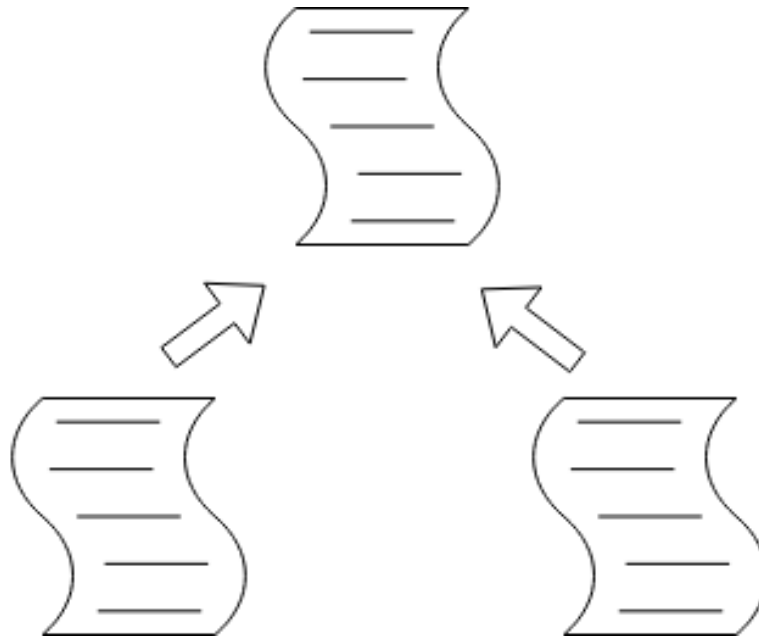
Gerência de Configuração

☐ Problema Exemplo:

- ☐ Agora seu colega de turma também quer editar o mesmo código
- ☐ Vocês baixam o arquivo
- ☐ Você edita e salva
- ☐ Seu colega edita e salva, sobrescrevendo seu código

O Versionamento do Código Resolve

- ☐ O controle de versão do código realiza o ‘merge’ das alterações



Versionamento de Código

- ☐ Controle de versão é uma sistema que mantém um registro das modificações
- ☐ Permite desenvolvimento colaborativo
- ☐ Permite saber quem fez as mudanças e quando
- ☐ **Permite reverter qualquer mudança e voltar para um estado anterior**

Ferramentas para Versionamento de Código

- ☐ Subversion (SVN)
- ☐ Mercurial
- ☐ CVS - Concurrent Versioning System
- ☐ Bazaar
- ☐ Git
 - ☐ Rápido, eficiente

Git

- ☐ Criado em 2005 por Linus Torvalds para auxiliar no desenvolvimento do kernel do Linux
- ☐ Como vimos, ele não é o único sistema de controle de versão, mas é o mais utilizado
- ☐ github.com
 - ☐ Serviço para armazenar repositório

Conceitos Básicos: Snapshot

- ☐ A forma que o git mantém o registro do histórico do seu código
- ☐ Registra como todos os seus arquivos são em um dado ponto no tempo
- ☐ Você decide quando fazer um snapshot, e de quais arquivos
- ☐ Poder voltar para visitar qualquer snapshot

Conceitos Básicos: Commit

- ☐ O ato de criar um snapshot
- ☐ Um projeto é essencialmente feito de vários commits
- ☐ Um commit contém três informações:
 - ☐ Informação de como o arquivo mudou comparado com anteriormente
 - ☐ Uma referência ao commit que veio antes
 - ☐ Um código hash

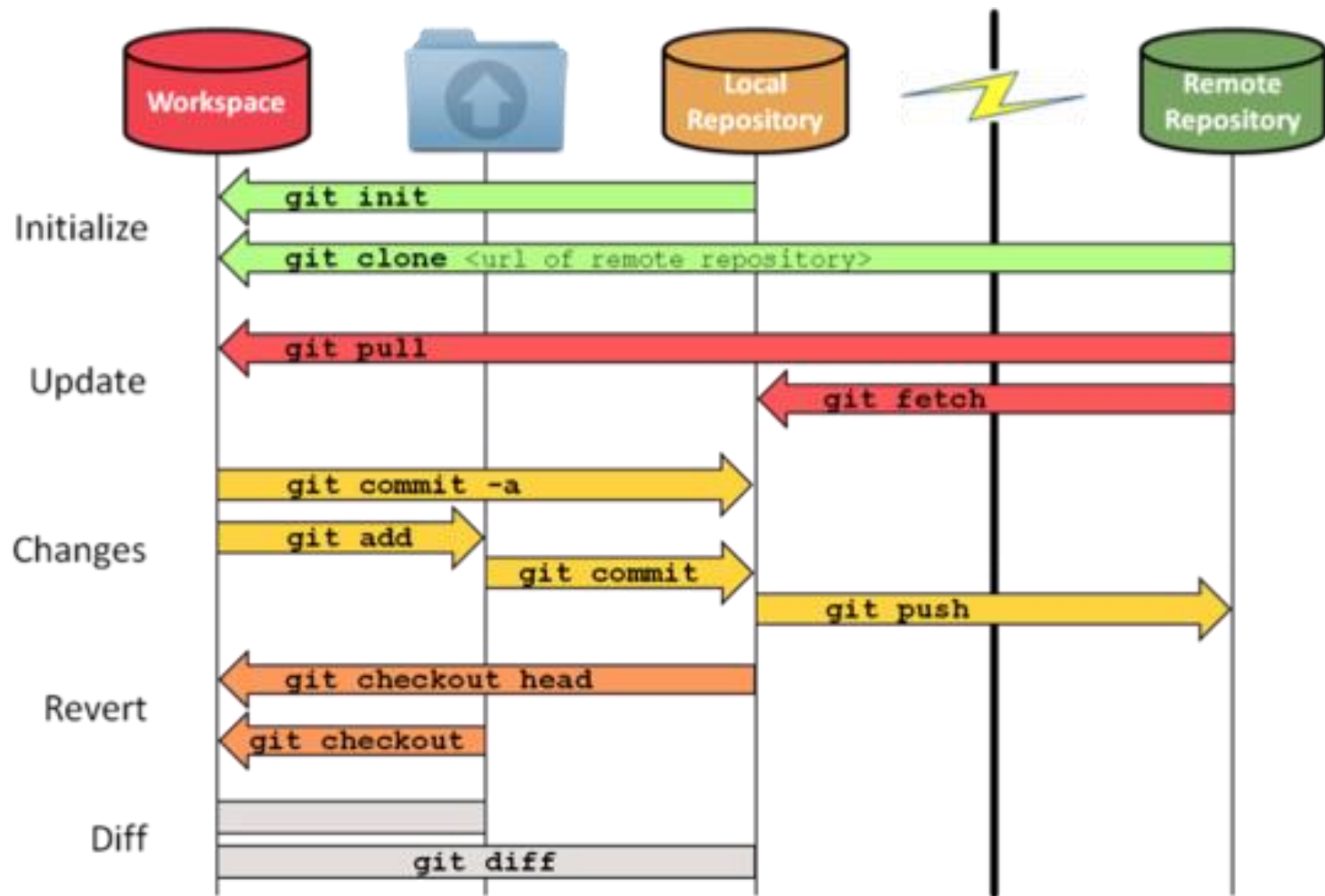
Conceitos Básicos: Repositório

- ☐ Frequentemente resumido para repo
- ☐ Uma coleção de arquivos e o histórico dos mesmos
 - ☐ Consiste de todos os seus commits
- ☐ Pode existir na máquina local ou em um servidor remoto (github)
- ☐ O ato de copiar um repositório de um servidor remoto é chamado clonagem (clone)

Conceitos Básicos: Repositório

- ☐ O ato de fazer o download de commits que não existem na sua máquina é chamado de pulling (pull)
- ☐ O processo de adicionar as suas mudanças locais no repositório remoto é chamado de pushing (push)

Exemplo Simples



Começando a usar

- ☐ Instalando o Git (<https://git-scm.com>)
 - ☐ `sudo apt-get install git`
- ☐ Windows:
 - ☐ <https://git-scm.com/download/win>
- ☐ Escolhendo sua interface gráfica <https://git-scm.com/downloads/guis>

Criação de conta no GitHub

- ☐ Acessar: <https://github.com/>
- ☐ Crie sua conta
- ☐ Se você é estudante não precisa pagar
 - ☐ <https://education.github.com/pack>
- ☐ Lembre-se que com esta conta você poderá contribuir com milhões de projetos open source

GitHub



Configurações Iniciais

- ☐ Conferindo sua versão
 - ☐ `$ git --version`
- ☐ Usuário
 - ☐ `$ git config --global user.name "Julio Reis"`
 - ☐ `$ git config --global user.email "julio.reis@dcc.ufmg.br"`
 - ☐ `$ git config --list`

Criando um Repositório

☐ Iniciando um repositório

- ☐ \$ cd project

- ☐ \$ git init

☐ Status

- ☐ \$ git status

☐ Adicionamento arquivos para versionamento

- ☐ \$ git add <file> ou git add . (para todos)

☐ Reset

- ☐ \$ git reset <file> ou git reset .

Operações

- ☐ Commit (consolidando/validando alterações feitas)

- ☐ \$ git commit -m "detailed message"

- ☐ Log

- ☐ \$ git log

- ☐ Show

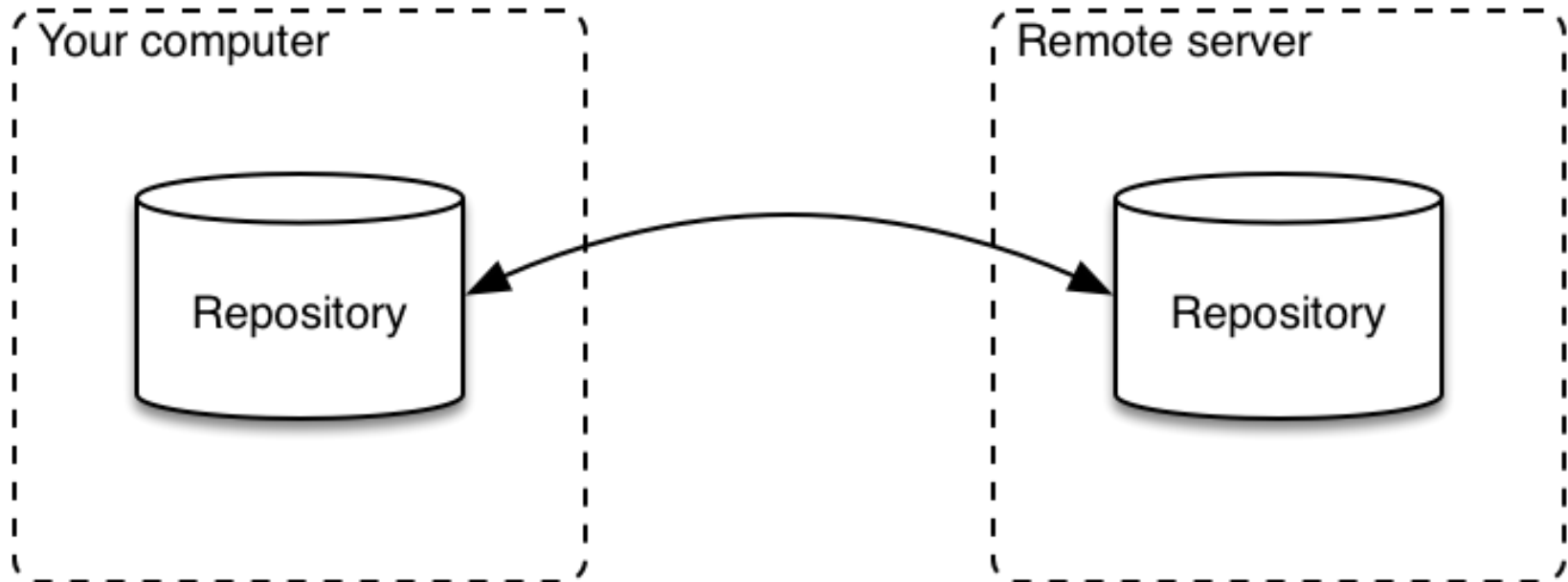
- ☐ \$ git show ou \$ git show <hash_id>

- ☐ Diff

- ☐ Working: \$ git diff ou git diff <file>

- ☐ Staging: \$ git diff --cached ou \$ git diff --cached <file>

Remote



Clone/Obtenção de Repositório

- ☐ \$ git clone [url]

- ☐ \$ git clone git@github.com:<user>/<project>.git

ou

- ☐ \$ git clone https://github.com/<user>/<project>.git
<folder>

- ☐ Verificando o status do seus arquivos:

 - ☐ \$ git status

- ☐ Monitorando novos arquivos:

 - ☐ \$ git touch new_file

 - ☐ \$ git add new_file

Outras Operações

- ☐ Ignorando arquivos

 - ☐ \$ touch .gitignore

- ☐ Log

 - ☐ \$ git log

- ☐ Show

 - ☐ \$ git show ou \$ git show <hash_id>

- ☐ Diff

 - ☐ Working: \$ git diff ou git diff <file>

 - ☐ Staging: \$ git diff --cached ou \$ git diff --cached <file>

Outras Operações

- ☐ Removendo arquivos

- ☐ \$ git rm <my_file>

- ☐ Movendo arquivos

- ☐ \$ git mv <my_file>

- ☐ Desfazendo operações:

- ☐ Modificando o último commit

- ☐ \$ git commit -amend

- ☐ Desfazendo arquivo modificado

- ☐ \$ git checkout <my_file>

- ☐ Removendo arquivo da área de validação

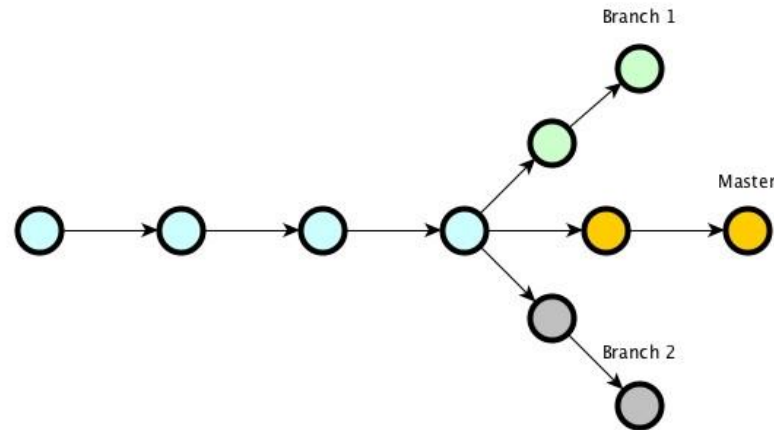
- ☐ \$ git reset HEAD <my_file>

Trabalhando com Remoto

- ☐ Criando repo a partir de um remoto
 - ☐ \$ git clone <url>
- ☐ Enviando alterações para o seu remoto
 - ☐ \$ git push origin master
- ☐ Recebendo alterações do seu remoto
 - ☐ \$ git pull origin master

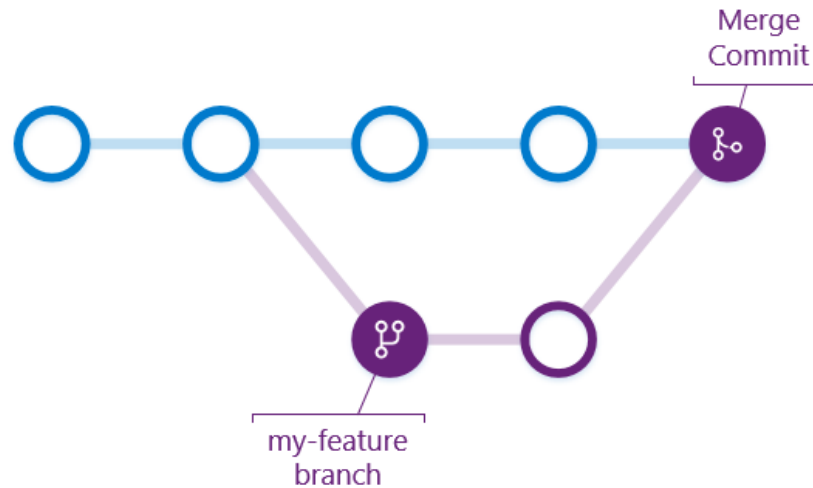
Branching

□ Uma feature = Uma branch



- ☐ \$ git branch my-feature
- ☐ \$ git checkout my-feature
- ☐ \$ git commit (x2)

Branching - Merge



- ☐ \$ git checkout master
- ☐ \$ git diff master..my-feature
- ☐ \$ git merge my-feature
- ☐ Clean-Up:
 - ☐ \$ git branch -d my-feature

Remote branch

☐ Local

- ☐ \$ git branch my-feature
- ☐ \$ git branch -d my-feature

☐ Remote

- ☐ \$ git branch -a
- ☐ \$ git push origin my-feature
- ☐ \$ git push origin --delete my-feature

Tagging

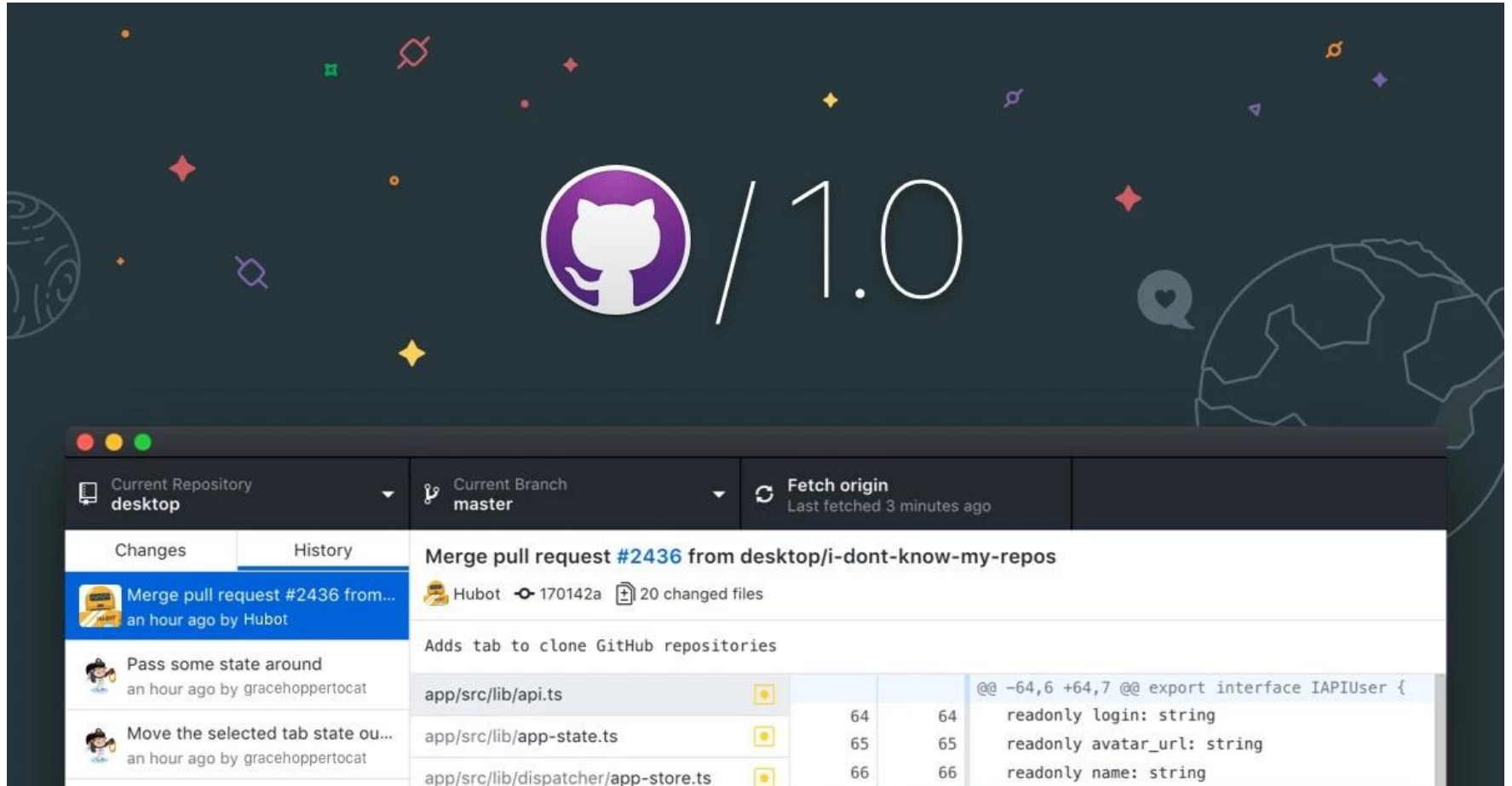
- ☐ O Git nos permite criar tags em pontos específicos do meu código (i.e. pontos importantes)
- ☐ Listar tags de um projeto
 - ☐ `$ git tag`
- ☐ Cria tags anotadas
 - ☐ `$ git tag -a v2.5 -m 'my_version2.5'`
- ☐ Exibe detalhes de uma tag
 - ☐ `$ git tag v2.5`
- ☐ Envia tags para o repositório remoto
 - ☐ `$ git push origin <tag_name>`

Git - Hooks

- ☐ Maneira de disparar scripts personalizados quando certas ações importantes acontecem
 - ☐ .git/hooks
 - ☐ pre-commit
 - ☐ git commit no-verify
 - ☐ prepare-commit-msg

GitHub Desktop

☐ desktop.github.com



Revisão de Código

Revisão de Código

- ❑ Sistemas grandes, complexos, é possível garantir...
 - ❑ Código legível? Sem duplicidade?
 - ❑ Facilidade de manutenção?
 - ❑ Ausência de erros?
- ❑ Taxa média de detecção de defeitos
 - ❑ Testes unitários: 25%
 - ❑ Testes de integração: 45%
- ❑ Conseguimos melhorar esses valores?

Revisão de Código

- ❑ Técnicas para “garantir” (e/ou melhorar) a qualidade do software desenvolvido
- ❑ Verificação
 - ❑ Software de acordo com a especificação
 - ❑ “Construímos o produto corretamente?”
- ❑ Validação
 - ❑ Software faz o que o usuário realmente deseja
 - ❑ “Construímos o produto certo (esperado)?”

Revisão de Código

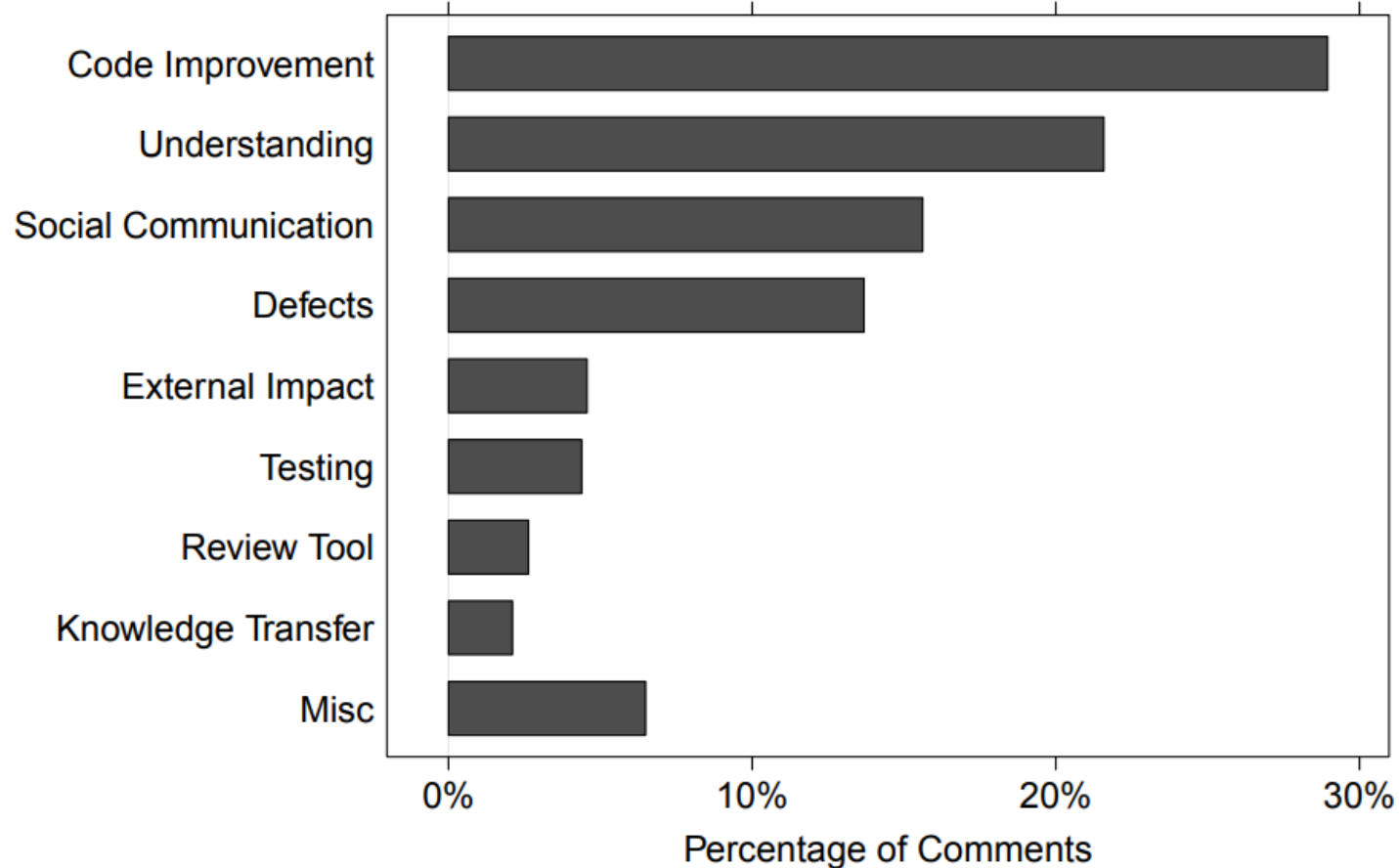
- Tarefa construtiva de rever o código e a documentação para identificar erros de interpretação, incoerências e outras falhas
 - Confirmação externa (antes de alterações e inserções de novos códigos)
- Propósito:
 - Melhorar o código
 - Melhorar o programador

Revisão de Código

Benefícios

- ❑ Taxa média de detecção de defeitos
 - ❑ Inspeções de design e código: 55% - 60%
- ❑ 11 programas desenvolvidos (mesma equipe):
 - ❑ 5 (sem revisões): 4,50 erros a cada 100 LoC
 - ❑ 6 (com revisões): 0,82 erros a cada 100 LoC
- ❑ Conhecimento:
 - ❑ Melhor entendimento do código
 - ❑ Feedback/Programadores Iniciantes

Revisão de Código



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

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

Revisão de Código @ Google

Modern Code Review: A Case Study at 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**

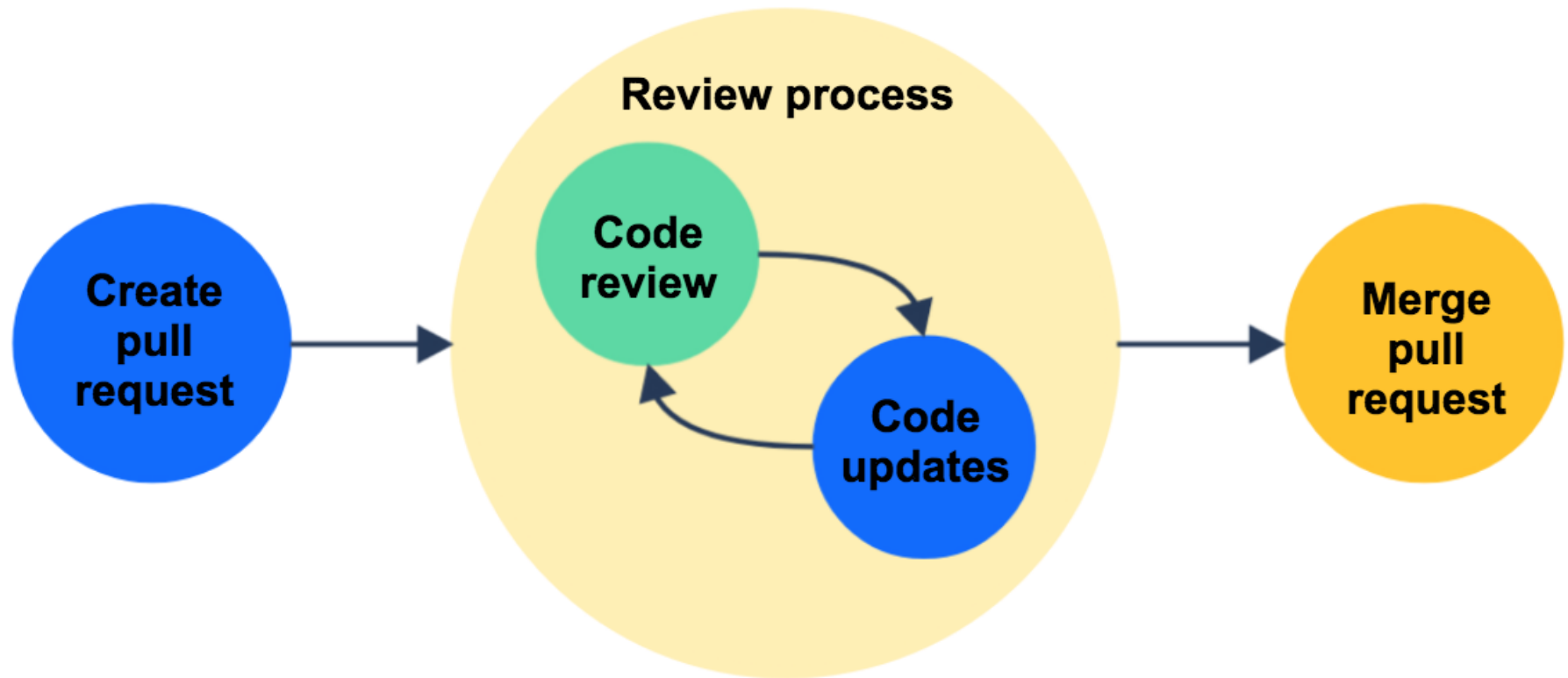
- ☐ Github

Revisão de Código

- Geralmente todo esse processo é feito antes de código ir para o repositório
 - Push no Github
- Série de ferramentas ajudam nesta tarefa

Revisão de Código

No Github



Revisão de Código

Checklist

| <i>Code Review Checklist</i> | |
|-------------------------------------|-------------------------------------|
| <input checked="" type="checkbox"/> | <u>Coding standards</u> |
| <input type="checkbox"/> | <u>Coding Best practices</u> |
| <input checked="" type="checkbox"/> | <u>Non Functional Requirements</u> |
| <input checked="" type="checkbox"/> | <u>OOAD Principles</u> |
| <input checked="" type="checkbox"/> | <u>Static Code Analysis Metrics</u> |
| <input type="checkbox"/> | <u>.....</u> |

Revisão de código

Boas Práticas

- ❑ Estabeleça metas quantificáveis para revisão
- ❑ Utilize listas de verificação (checklists)
- ❑ Registre e verifique se os defeitos são consertados
- ❑ Você não irá revisar tudo de uma vez!
 - ❑ Talvez será adequado priorizar partes críticas

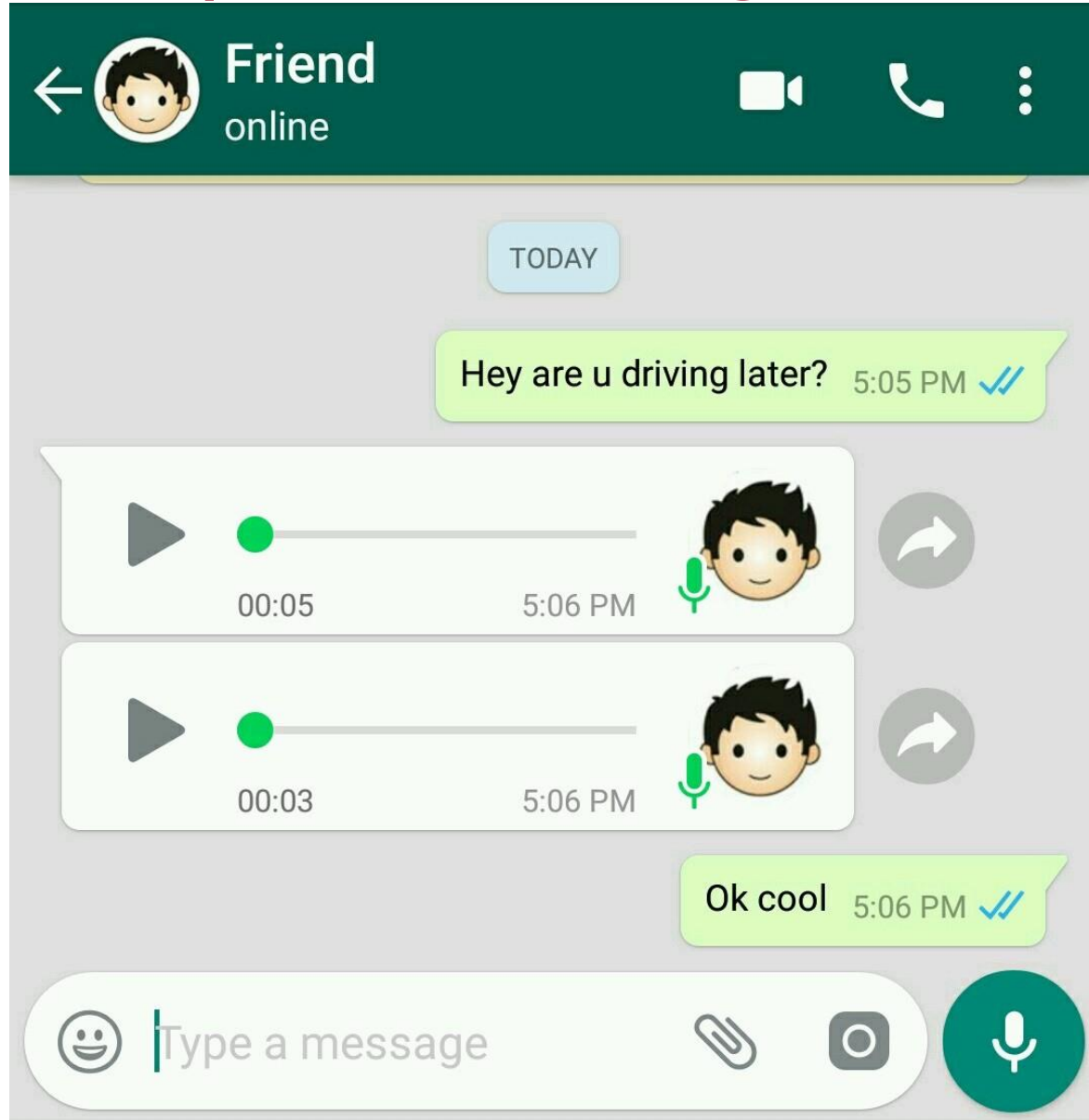
Programação e Desenvolvimento de Software 2

Interfaces e Polimorfismo

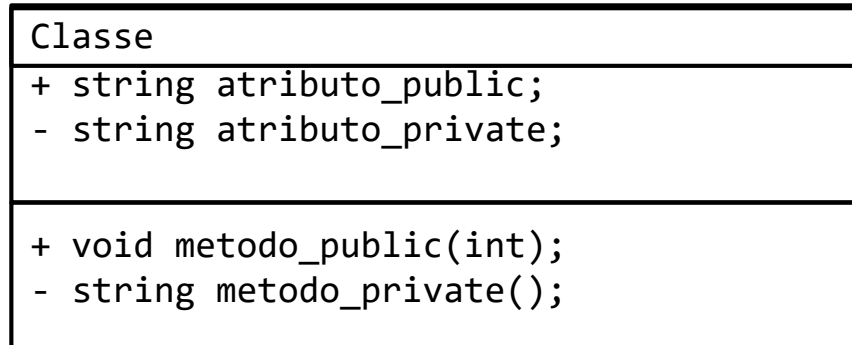
Prof. Julio Cesar S. dos Reis
julio.reis@dcc.ufmg.br

Interfaces

Diferentes tipos de Mensagens

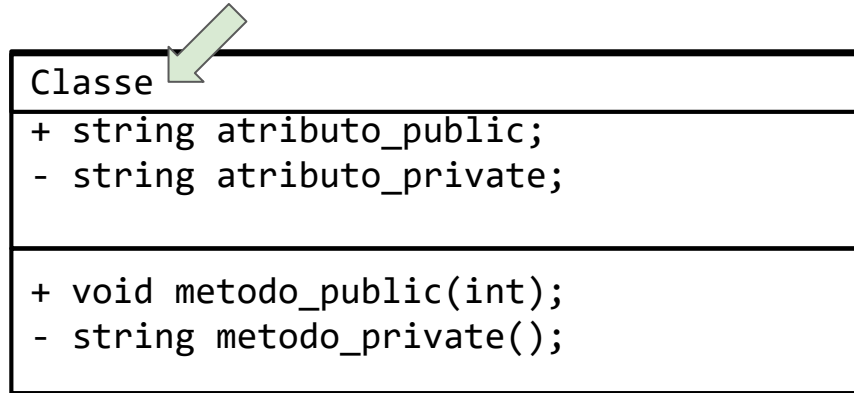


Unified Modelling Language



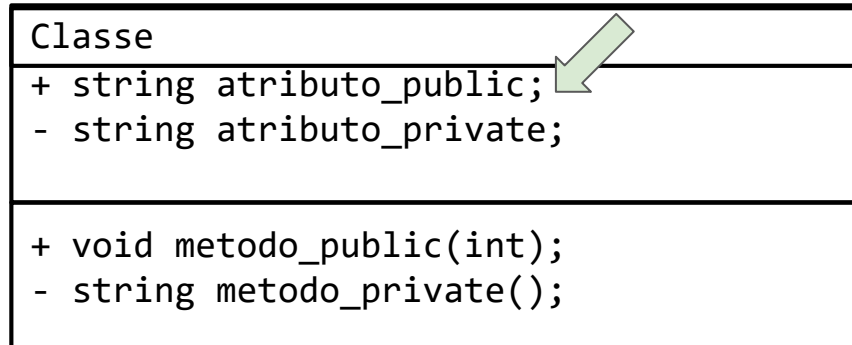
- ❑ UML define uma padrão de diagramas
- ❑ Úteis para o resto da disciplina

Unified Modelling Language



□ Nome da Classe

Unified Modelling Language

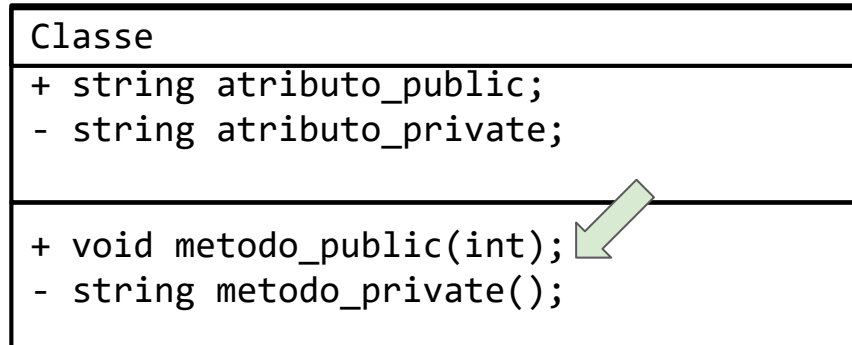


☐ Atributos

☐ + → public

☐ - → private

Unified Modelling Language

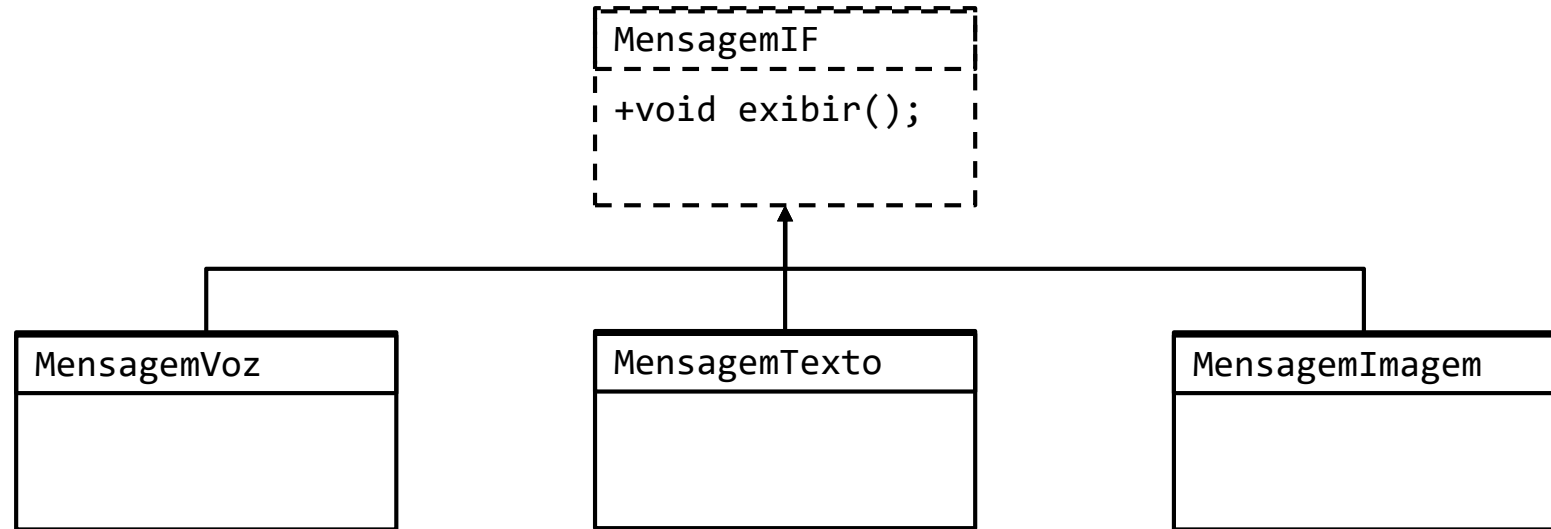


□ Métodos

□ + → public

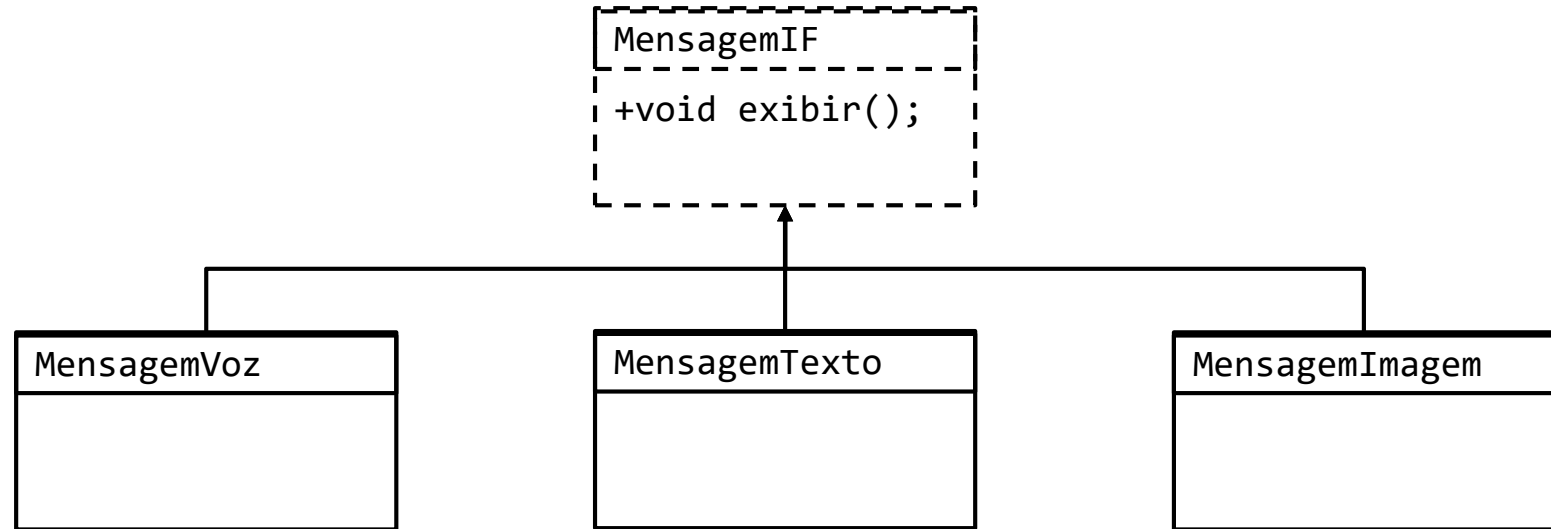
□ - → private

Unified Modelling Language



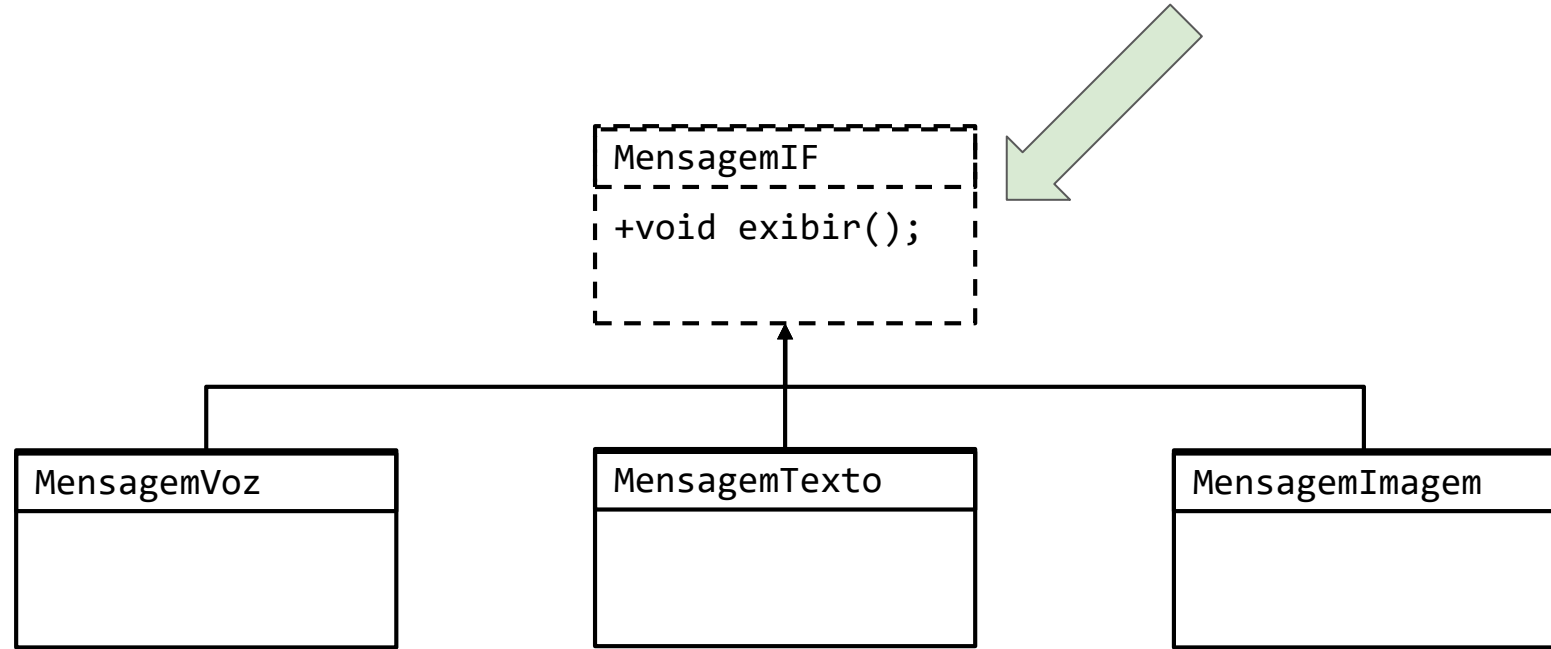
☐ Entendendo o diagrama:

Unified Modelling Language



- ☐ As **classes** ajudam a definir um objeto e seu comportamento e as **interfaces** que auxiliam na definição dessas classes
- ☐ Declaração (assinatura) de métodos

Unified Modelling Language

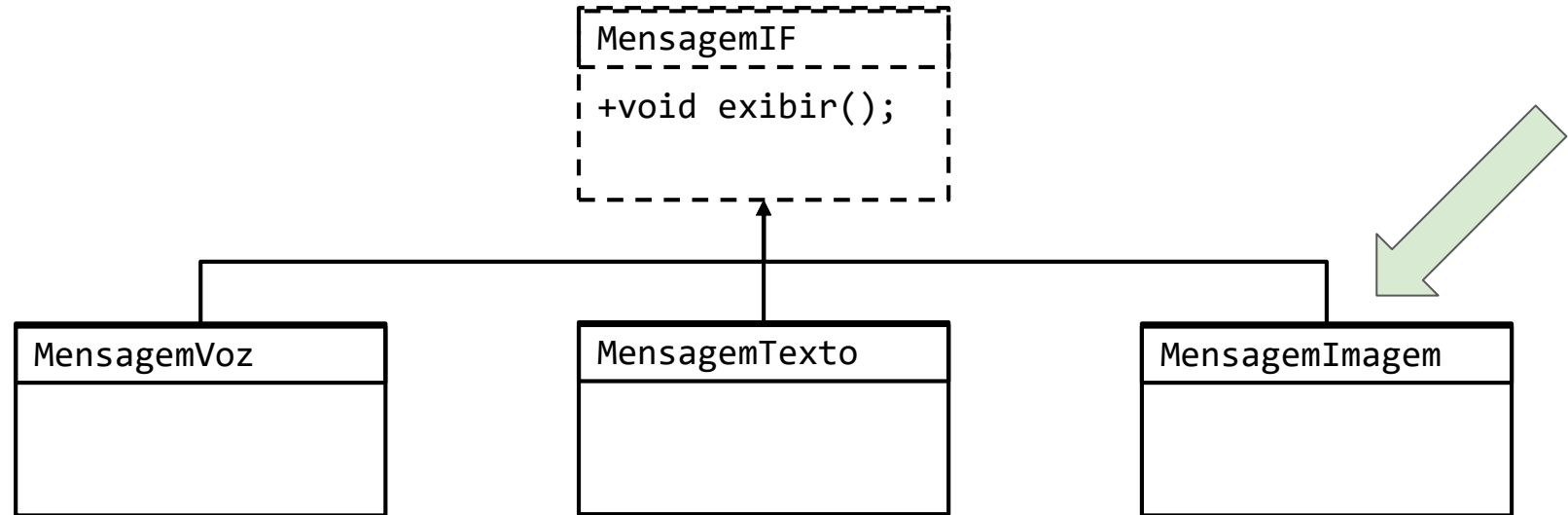


☐ Interfaces

☐ Topo da hierarquia de mensagens

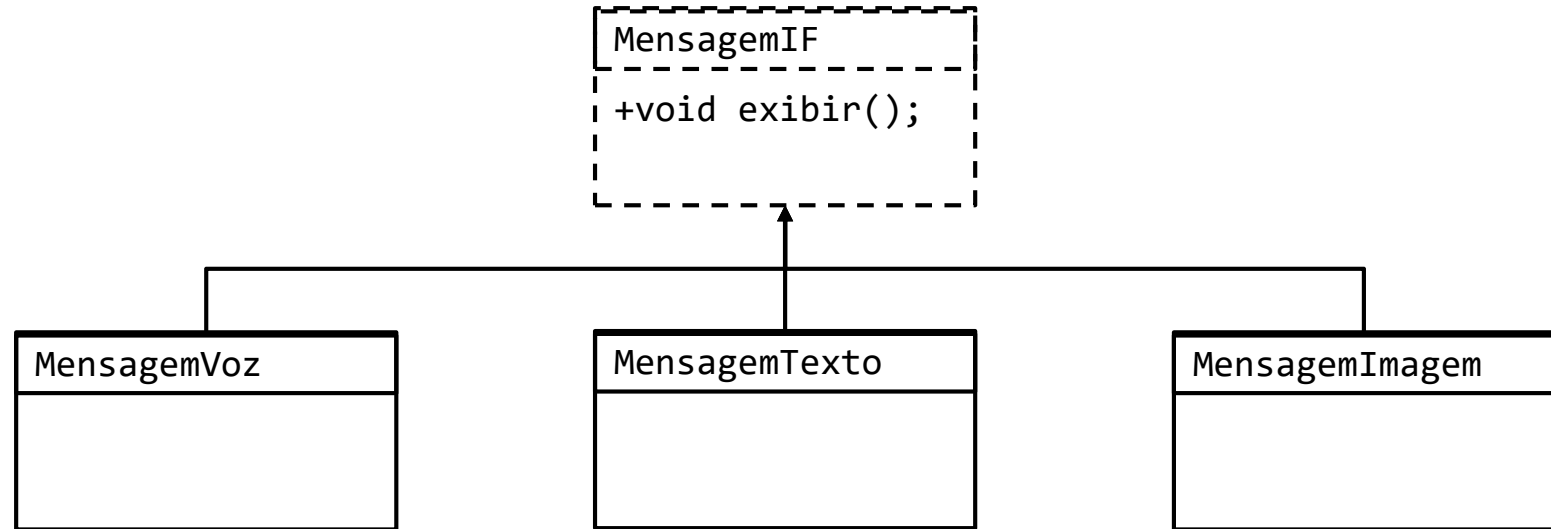
☐ Pontilhada pois nunca é implementada

Unified Modelling Language



- ☐ Classes:
 - ☐ Já conhecemos elas
 - ☐ Definem um comportamento comum

Comportamento Padrão



- ☐ Como esses objetos devem responder ao receberem o mesmo sinal: exibir?
 - ☐ Todos respondem da mesma forma?
- ☐ Vai existir um comportamento padrão?

Polimorfismo

- ❑ Termo originário do grego
 - ❑ Poli: muitas
 - ❑ Morphos: formas
- ❑ POO
 - ❑ Objetos de classes diferentes responderem a uma mesma mensagem de diferentes maneiras
 - ❑ Várias formas de responder à mensagem

Polimorfismo

- Utilizar um mesmo nome para se referir a diferentes métodos sobre um certo tipo
 - Objeto decide qual método deve ser
- Exemplo
 - Hierarquia de mensagens
 - Classe mais genérica possui o método **exibir**

Polimorfismo

- Programação voltada a tipos abstratos
- Possibilidade de um tipo abstrato (classe abstrata ou interface) ser utilizado sem que se conheça a implementação concreta
 - Independência de implementação
 - Maior foco na interface (fronteira, contrato)

Interfaces

- Definidas com métodos virtuais
- Não podem ser instanciadas
 - virtual = 0 garantem isso

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H

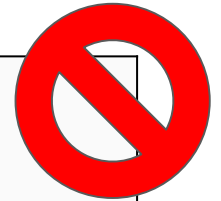
class MensagemIF {
public:
    virtual void exhibir() = 0;
};

#endif
```

Interfaces

- ❑ Não podem ser instanciadas
 - ❑ 2 linhas com erro abaixo

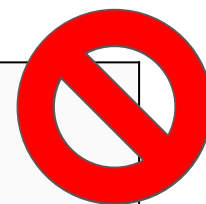
```
int main(void) {  
    MensagemIF msg = MensagemIF();  
    MensagemIF *msg2 = new MensagemIF();  
    MensagemIF msg3;  
}
```



Interfaces

□ Como fazer uso?!

```
int main(void) {  
    MensagemIF msg = MensagemIF();  
    MensagemIF *msg2 = new MensagemIF();  
    MensagemIF msg3;  
}
```



Implementando Interfaces

Definimos o comportamento nas classes

```
#ifndef PDS2_MENSAGEMTEXT0_H
#define PDS2_MENSAGEMTEXT0_H

#include <string>
#include "mensagem.h"

class MensagemTexto : public MensagemIF {
private:
    std::string _msg;
public:
    MensagemTexto(std::string msg);
    virtual void exhibir();
};

#endif
```

Implementando Interfaces

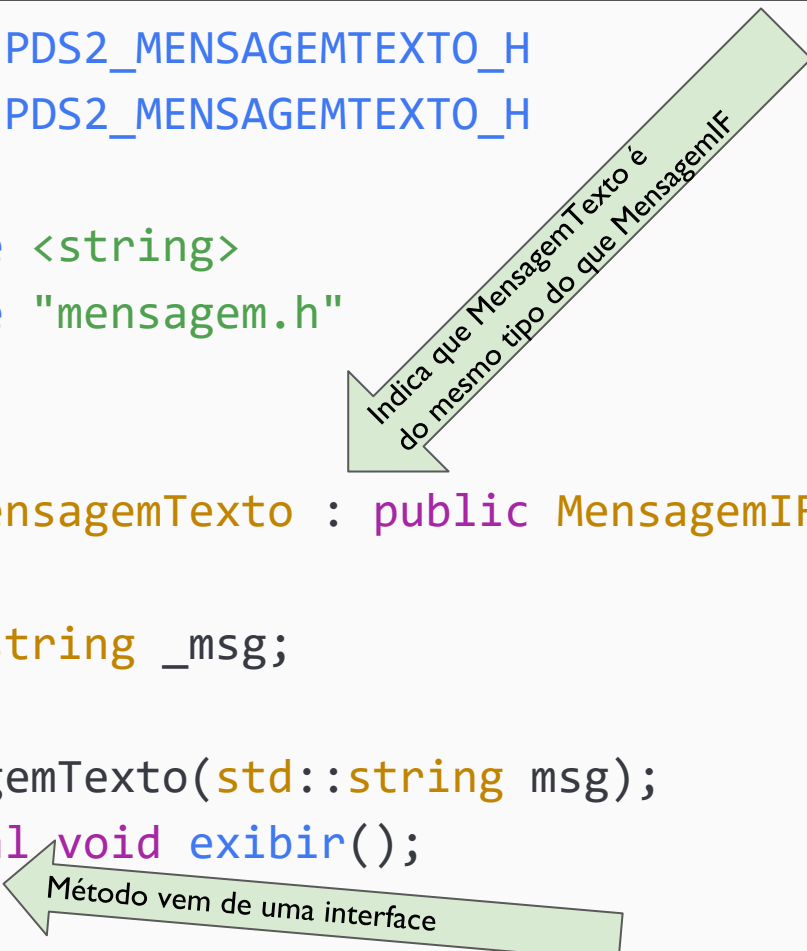
Definimos o comportamento nas classes

```
#ifndef PDS2_MENSAGEMTEXTTO_H
#define PDS2_MENSAGEMTEXTTO_H

#include <string>
#include "mensagem.h"

class MensagemTexto : public MensagemIF {
private:
    std::string _msg;
public:
    MensagemTexto(std::string msg);
    virtual void exhibir();
};

#endif
```



Indica que MensagemTexto é do mesmo tipo do que MensagemIF

Método vem de uma interface

Implementando Interfaces

Podemos definir uma mensagem com imagens

```
#ifndef PDS2_MENSAGEMIMG_H  
#define PDS2_MENSAGEMIMG_H
```

```
#include <string>  
#include "mensagem.h"
```

Indica que MensagemTexto é
do mesmo tipo do que MensagemIF

```
class MensagemImagem : public MensagemIF {  
private:  
    std::string _arquivo;  
public:  
    MensagemTexto(std::string arquivo);  
    virtual void exhibir();  
};
```

Método vem de uma interface

```
#endif
```

Implementando

Mais de um tipo de mensagem

```
#include "mensagemtexto.h"

#include <iostream>

MensagemTexto::MensagemTexto(std::string msg) {
    this->_msg = msg;
}

void MensagemTexto::exibir() {
    std::cout << this->_msg;
    std::cout << std::endl;
}
```

```
#include "mensagemimg.h"

#include <fstream>
#include <iostream>

MensagemImagem::MensagemImagem(std::string arquivo) {
    this->_arquivo = arquivo;
}

void MensagemImagem::exibir() {
    std::ifstream arquivo(this->_arquivo);
    std::string line;
    while (std::getline(arquivo, line))
        std::cout << line << std::endl;
    arquivo.close();
}
```

Implementando

Mais de um tipo de mensagem

```
#include "mensagemtexto.h"
```

```
#include <iostream>
```

```
MensagemTexto::MensagemTexto(std::string msg) {  
    this->_msg = msg;  
}
```

```
void MensagemTexto::exibir() {  
    std::cout << this->_msg;  
    std::cout << std::endl;  
}
```

Exibe um texto

```
#include "mensagemimg.h"
```

```
#include <fstream>
```

```
#include <iostream>
```

```
MensagemImagem::MensagemImagem(std::string arquivo) {  
    this->_arquivo = arquivo;  
}
```

```
void MensagemImagem::exibir() {  
    std::ifstream arquivo(this->_arquivo);  
    std::string line;  
    while (std::getline(arquivo, line))  
        std::cout << line << std::endl;  
    arquivo.close();  
}
```

Exibe uma imagem ascii

Polimorfismo em ação

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

Polimorfismo em ação

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
```


Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
Tocando o arquivo... audio.wav
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
Tocando o arquivo... audio.wav
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
Tocando o arquivo... audio.wav
```

Qual o tipo?!

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```

```
$ ./main
Oi, tem aula de PDS2 hoje?
Tocando o arquivo... audio.wav

      /      \
    :  o    o  ;
   (          (_ )
  :              ;
   \      _      /
  \_ ._____ .-'
   /`""""`\
      / , \
     /|/\|/\| _\
    (_|/\|/\|/\|_)
           |_____|
          _)_ _ | _ ( _
         (____|____)
```

Erros Comuns

- ❑ Tentar usar o tipo genérico na declaração
- ❑ Erro de compilação
 - ❑ Tipos com tamanhos diferentes (assinatura)



```
void exibir_na_tela(MensagemIF &msg) {  
    msg.exibir();  
}  
  
int main(void) {  
    MensagemIF texto = MensagemTexto("Oi, tem aula de PDS2 hoje?");  
    MensagemIF audio = MensagemVoz("audio.wav");  
  
    exibir_na_tela(texto);  
    exibir_na_tela(audio);  
}
```

Solução (se necessário)

□ Ponteiros

□ Sempre tem um tamanho fixo

```
#include "mensagem.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF *msg) {
    msg->exibir();
}

int main(void) {
    MensagemIF *texto = new MensagemTexto("Oi, tem aula de PDS2 hoje?");
    MensagemIF *audio = new MensagemVoz("audio.wav");
    exibir_na_tela(texto);
    exibir_na_tela(audio);
    delete texto;
    delete audio;
}
```

Exercício I



Exercício I

```
class Animal
{
    public:
        virtual void fale();
};
```

```
class Gato : public Animal
{
    public:
        void fale() override {
            cout << "Miau!"
<< endl;
        }
};
```

```
class Cachorro : public Animal
{
    public:
        void fale() override {
            cout << "Au!Au!"
<< endl;
        }
};
```


Exercício I

```
int main()
{
    Cachorro c;
    c.fale();

    Gato g;
    g.fale();

    return 0;
};
```

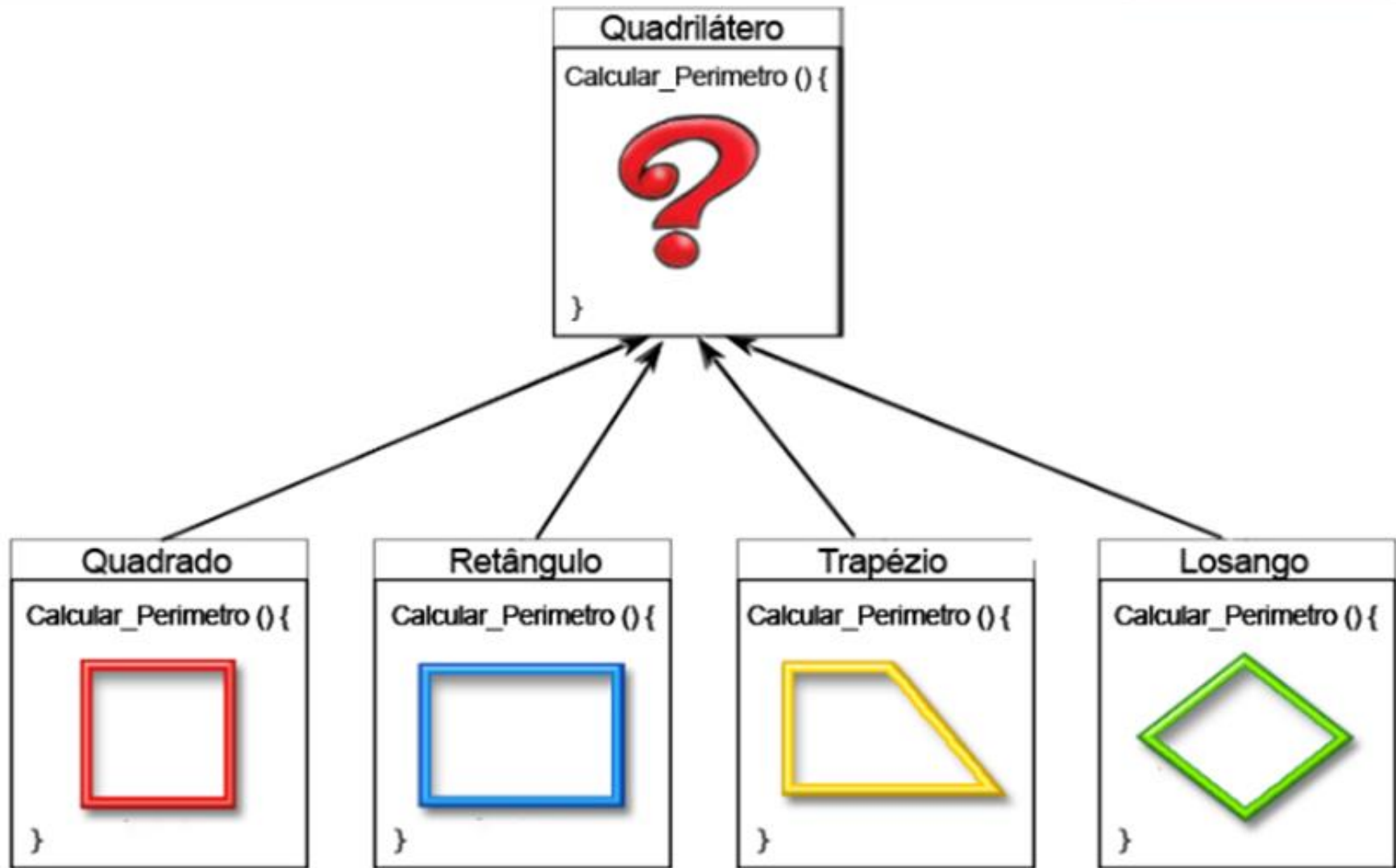
Exercício I

```
int main()
{
    Animal * c = new Cachorro();
    c -> fale();
    delete c;

    Animal * g = new Gato();
    g -> fale();
    delete g;

    return 0;
};
```

Exercício 2

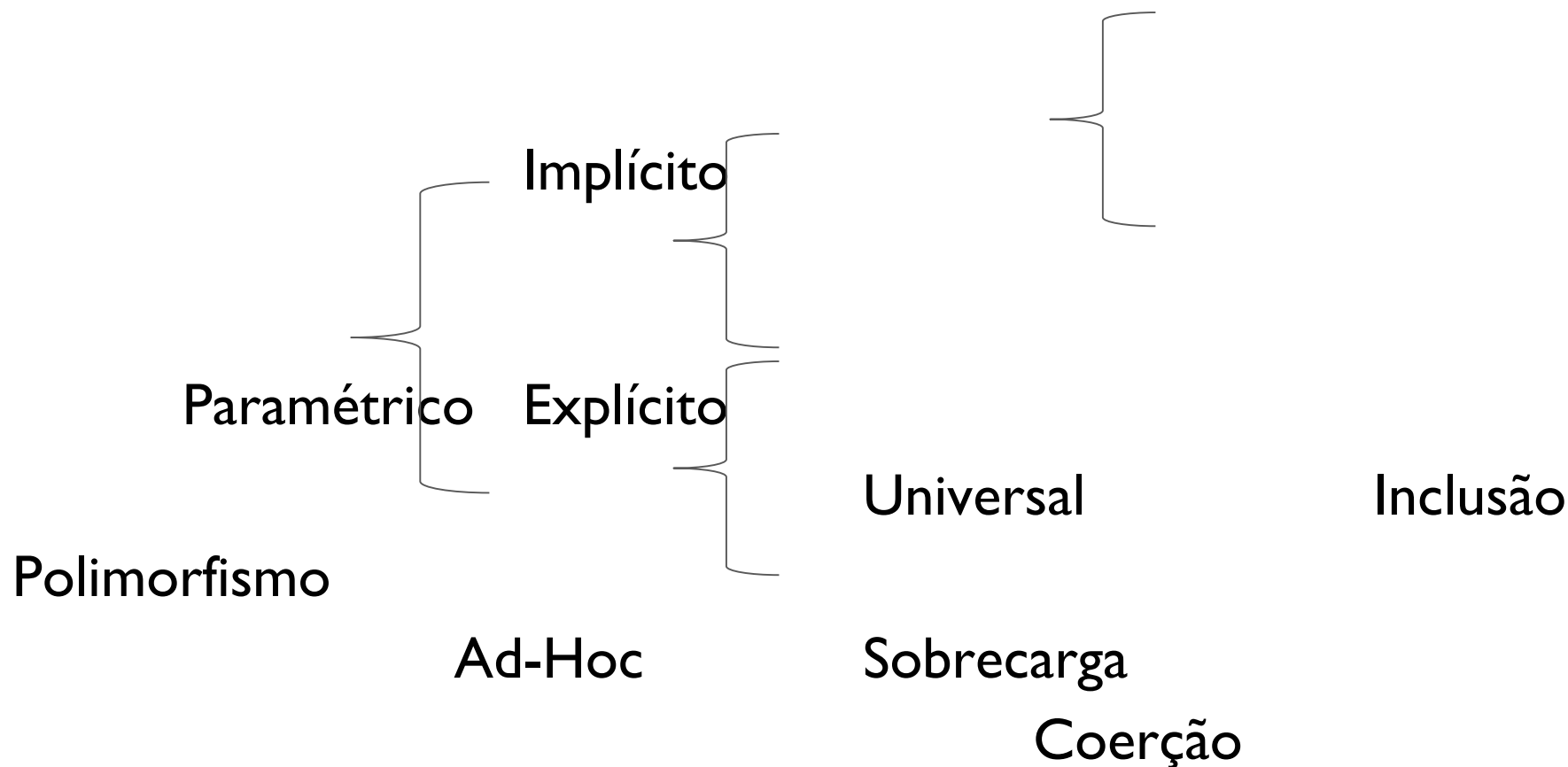


Tipos de Polimorfismo

Polimorfismo

- Seleção da instância (forma) do objeto
 - Ligação Prematura (Early binding)
 - As decisões são feitas durante a compilação
 - Ligação Tardia (Late binding)
 - As decisões são feitas durante a execução
 - É a chave para o funcionamento do polimorfismo
- C++ \Rightarrow Padrão é ligação prematura
 - Ligação tardia utiliza o comando “virtual”

Polimorfismo



Tipos de Polimorfismo

Universal

- Universal ou Verdadeiro
 - Quando uma função ou tipo trabalha de maneira uniforme para uma gama de tipos definidos na linguagem
- A mesma definição (código) de uma função pode ser utilizada por diferentes tipos
- Potencialmente número infinito de variações

Tipos de Polimorfismo

Universal Paramétrico

- Torna a linguagem mais expressiva
 - Templates em C++
- Universal paramétrico
 - Os tipos são identificados pelo compilador
 - São passados implicitamente à função

Tipos de Polimorfismo

Early Binding

```
#include <list>
```

```
int main() {
```

```
    std::list<Pessoa> lista;
```

```
    Pessoa p;
```

```
    lista.push_back(p);
```

```
    std::cout << lista.size() << std::endl;
```

```
    return 0;
```

```
}
```

Template →
Universal Paramétrico

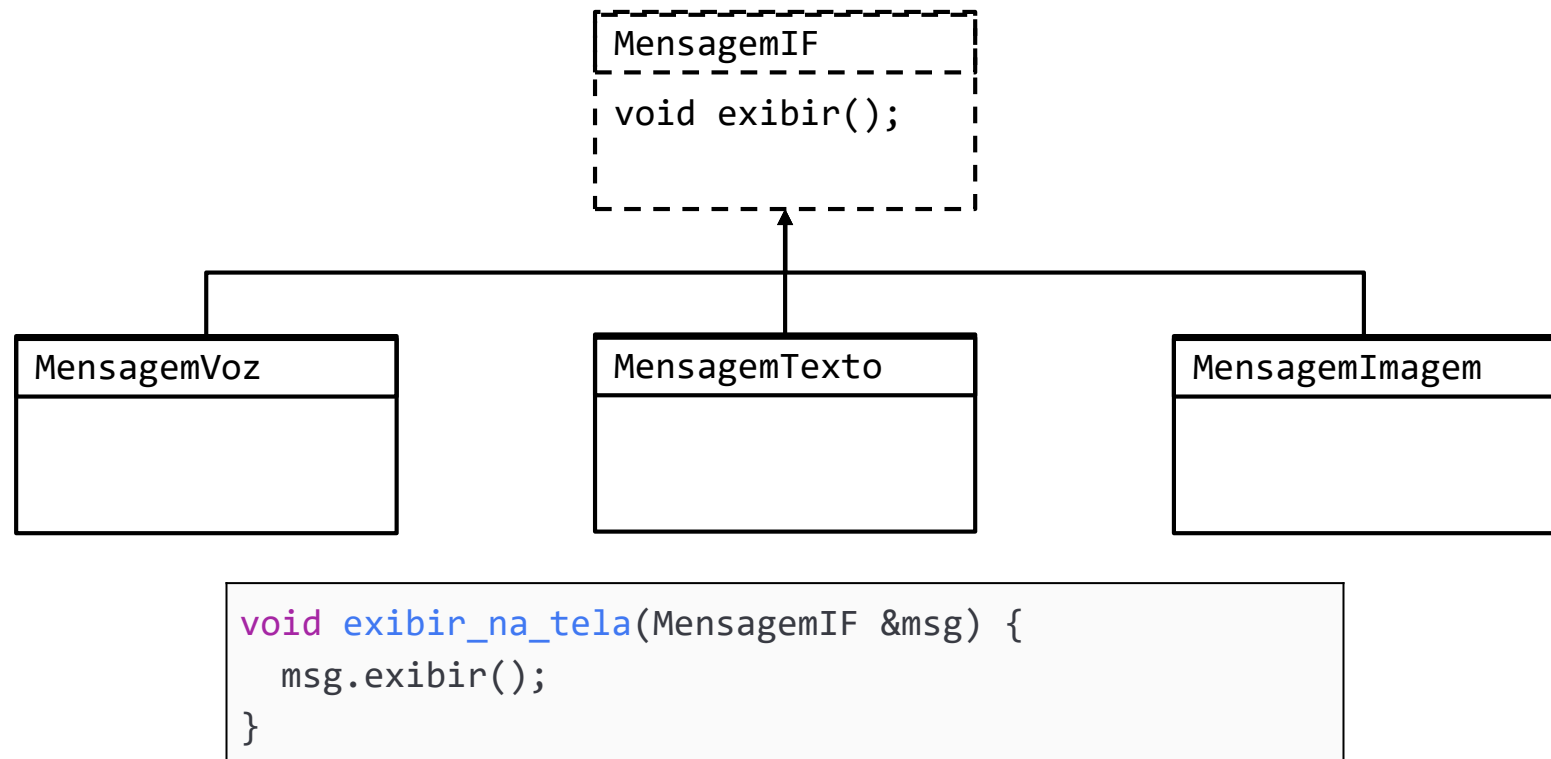
Tipos de Polimorfismo

Late Binding (Universal - Inclusão)

- Modela subtipos
 - Redefinição em classes descendentes
 - O subtipo está incluído no próprio tipo
- Onde um objeto de um tipo for esperado, um objeto do subtipo deve ser aceito
- Princípio da substituição de Liskov
 - se S é um subtipo de T, então os objetos do tipo T, em um programa, podem ser substituídos pelos objetos de tipo S sem que seja necessário alterar as propriedades deste programa.
- O contrário nem sempre é válido!

Tipos de Polimorfismo

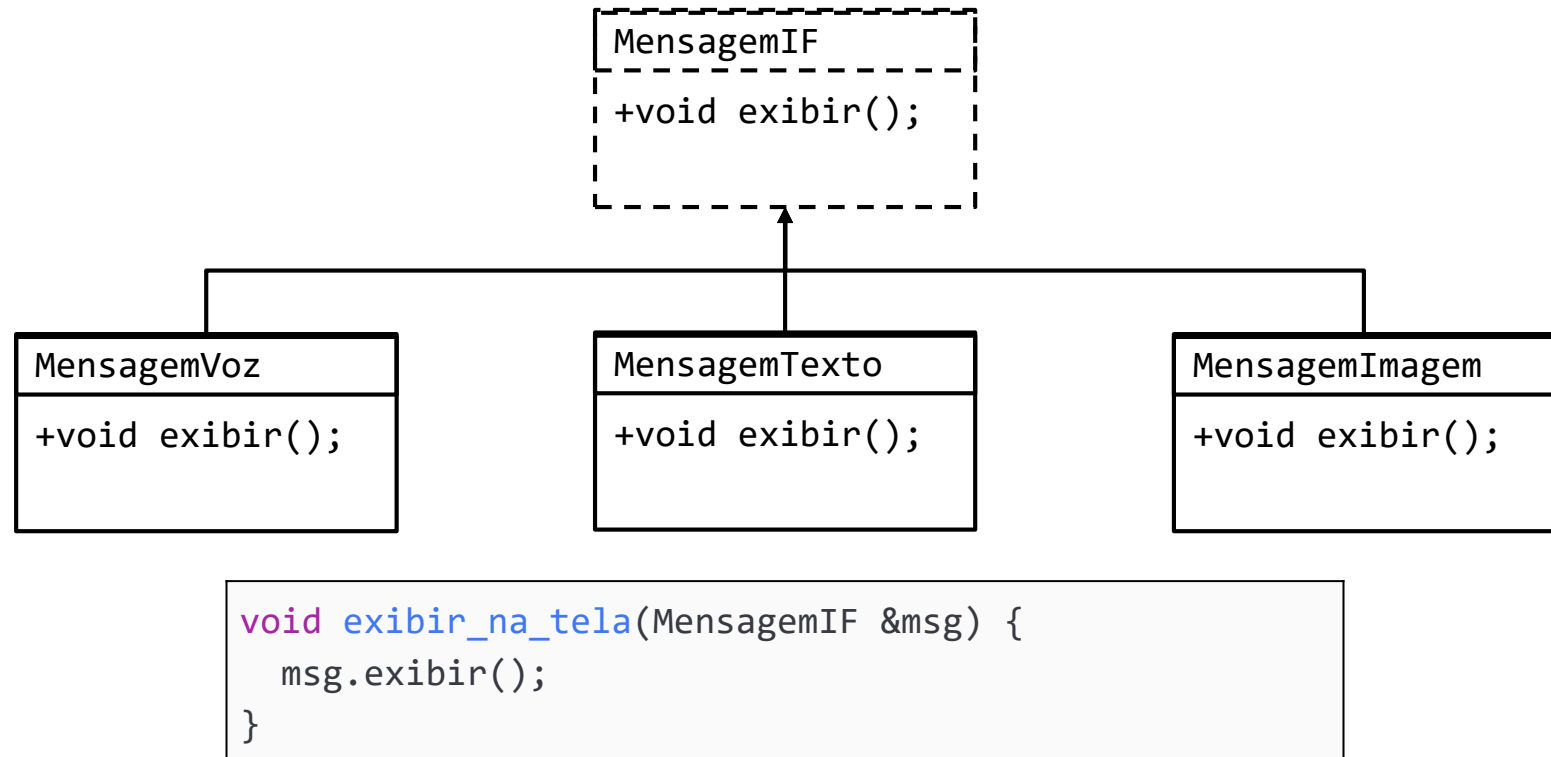
Universal - Inclusão



- ❑ Podemos passar qualquer subtipo de `MensagemIF` para o método acima

Tipos de Polimorfismo

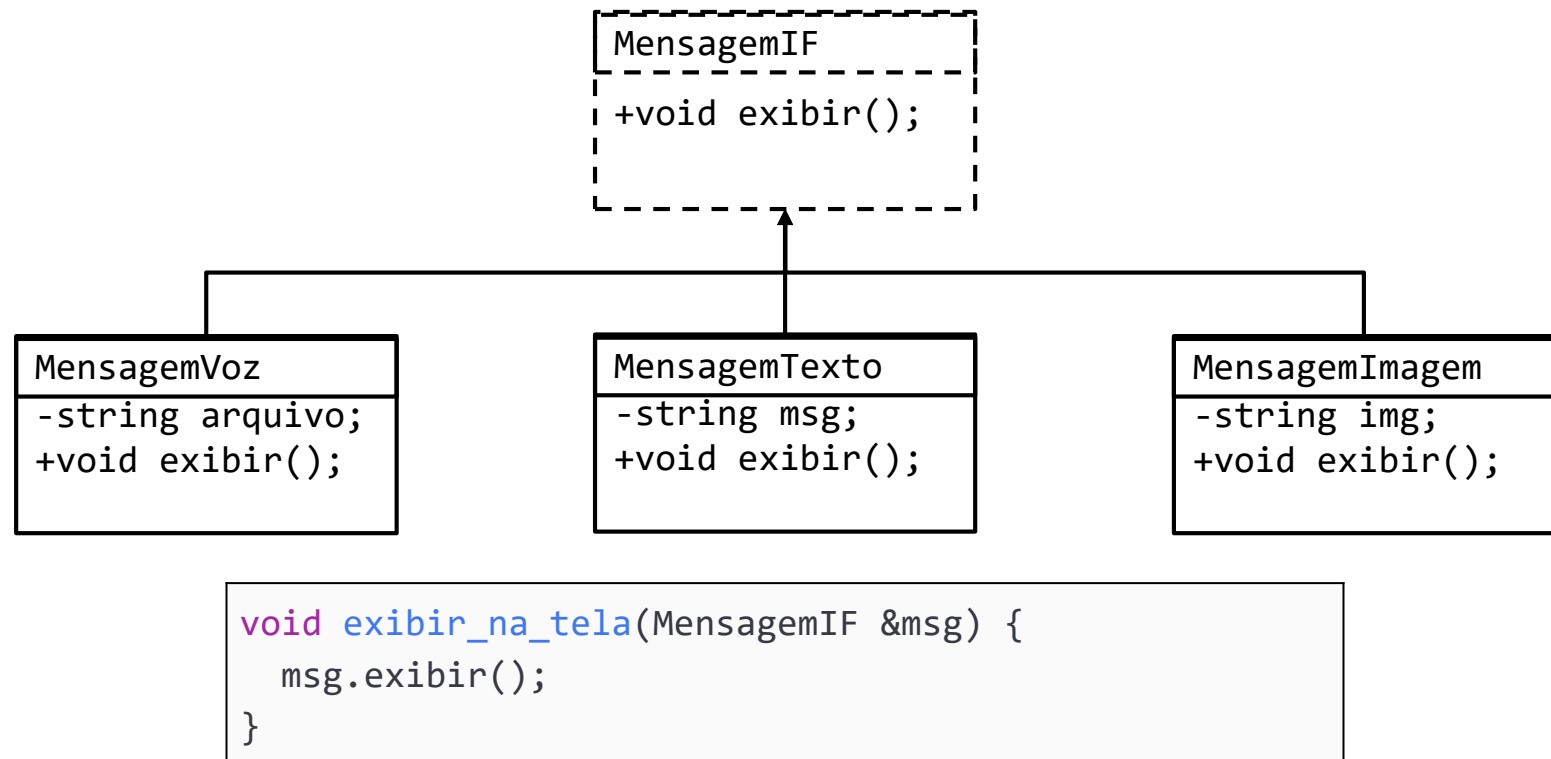
Universal - Inclusão



□ Todas suportam o exibir

Tipos de Polimorfismo

Universal - Inclusão

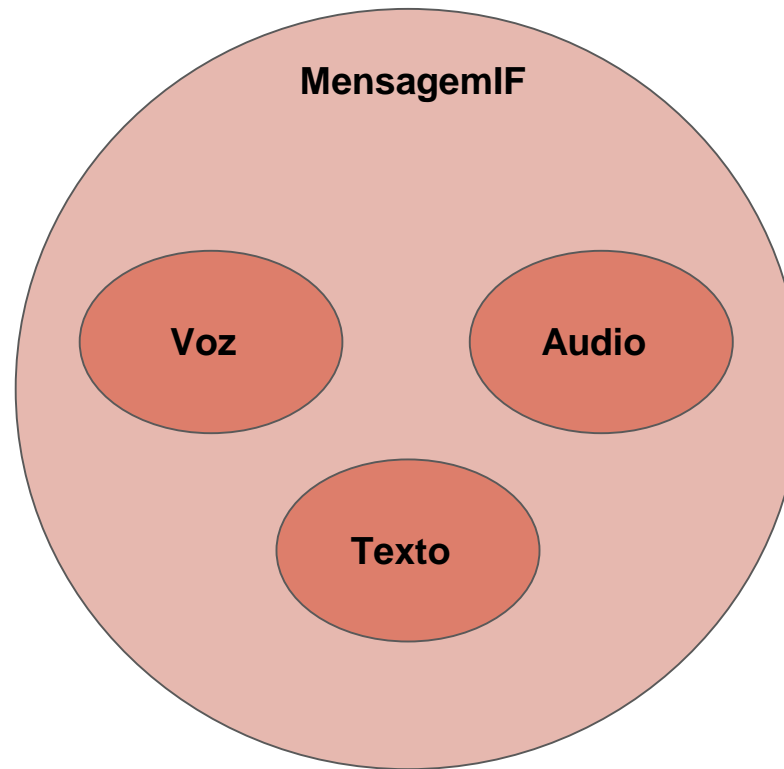


- Porém cada uma tem comportamento interno (private) diferente

Tipos de Polimorfismo

Inclusão

Contexto de Tipos



Tipos de Polimorfismo

Ad-hoc - Sobrecarga

- Número finito de entidades distintas, todas com mesmo nome, mas códigos distintos
- Função ou valor conforme o contexto

Tipos de Polimorfismo

Ad-hoc - Sobrecarga

- O mesmo identificador denota diferentes funções que operam sobre tipos distintos
- Resolvido estaticamente (compilação)
 - Considera os tipos para escolher a definição
 - Difere no número e no tipo dos parâmetros

Tipos de Polimorfismo

Ad-hoc - Sobrecarga

```
class Ponto {  
private:  
    double _x = 0;  
    double _y = 0;  
public:  
    void set_xy(double x, double y) {  
        this->_x = x;  
        this->_y = y;  
    }  
  
    void set_xy(double xy) {  
        this->_x = xy;  
        this->_y = xy;  
    }  
};
```

Tipos de Polimorfismo

Ad-hoc - Coerção

- Conversão automática de tipo
 - Utilizada para satisfazer o contexto atual
 - Considera a definição para escolher o tipo
- Linguagem possui um mapeamento interno

Conversão de Tipos (Casting)

Caso de Estudo 2

Coleções

```
#ifndef PDS2_LISTADUPLA_H
#define PDS2_LISTADUPLA_H

struct node_t {
    int elemento;
    node_t *anterior;
    node_t *proximo;
};

class ListaDuplamenteEncadeada {
private:
    node_t *_inicio;
    node_t *_fim;
    int _num_elementos_inseridos;
public:
    ListaDuplamenteEncadeada();
    ~ListaDuplamenteEncadeada();
    void inserir_elemento(int elemento);
    void imprimir();
    int tamanho();
    void remove_iesimo(int i);
};
#endif
```

```
#ifndef PDS2_BST_H
#define PDS2_BST_H

#include "node.h"

class BST {
private:
    Node *_raiz;
    int _num_elementos_inseridos;
public:
    BST();
    ~BST();
    void inserir_elemento(int elemento);
    void imprimir();
    int tamanho();
    bool tem_elemento(int elemento);
};

#endif
```

Caso de Estudo 2

Note três comportamentos repetidos: insere, imprime, tamanho

```
#ifndef PDS2_LISTADUPLA_H
#define PDS2_LISTADUPLA_H

struct node_t {
    int elemento;
    node_t *anterior;
    node_t *proximo;
};

class ListaDuplamenteEncadeada {
private:
    node_t *_inicio;
    node_t *_fim;
    int _num_elementos_inseridos;
public:
    ListaDuplamenteEncadeada();
    ~ListaDuplamenteEncadeada();
    void inserir_elemento(int elemento);
    void imprimir();
    int tamanho();
    void remove_iesimo(int i);
};
#endif
```

```
#ifndef PDS2_BST_H
#define PDS2_BST_H

#include "node.h"

class BST {
private:
    Node *_raiz;
    int _num_elementos_inseridos;
public:
    BST();
    ~BST();
    void inserir_elemento(int elemento);
    void imprimir();
    int tamanho();
    bool tem_elemento(int elemento);
};

#endif
```

Agregando o comportamento similar

Fazendo uso de interfaces

□ Note o destrutor virtual

```
#ifndef PDS2_COLECAO_H
#define PDS2_COLECAO_H

class ColecaoIF {
public:
    virtual ~ColecaoIF() {};
    virtual void inserir_elemento(int elemento) = 0;
    virtual void imprimir() = 0;
    virtual int tamanho() = 0;
};

#endif
```

Agregando o comportamento similar

Fazendo uso de interfaces

- Note o destrutor virtual
 - Às vezes precisamos chamar o destrutor quando temos um tipo da interface

```
#ifndef PDS2_COLECAO_H
#define PDS2_COLECAO_H
class ColecaoIF {
public:
    virtual ~ColecaoIF() {};
    virtual void inserir_elemento(int elemento) = 0;
    virtual void imprimir() = 0;
    virtual int tamanho() = 0;
};
#endif
```

Pulando para o main

Uso do destrutor virtual

```
#include "colecacao.h"
#include "listadupla.h"
#include "bst.h"

int main(void) {
    ColecaoIF *lista = new ListaDuplamenteEncadeada();
    lista->inserir_elemento(2);
    lista->inserir_elemento(3);

    ColecaoIF *bst = new BST();
    bst->inserir_elemento(10);
    bst->inserir_elemento(-1);
    bst->inserir_elemento(6);

    lista->imprimir();
    bst->imprimir();

    delete lista;
    delete bst;
    return 0;
}
```



lista e bst são ColecaoIF

Agregando o comportamento similar

Fazendo uso de interfaces

- O destrutor virtual é um código sem nada
 - Podemos implementar o mesmo caso tenha um comportamento comum (isso é raro)

```
#ifndef PDS2_COLECAO_H
#define PDS2_COLECAO_H
class ColecaoIF {
public:
    virtual ~ColecaoIF() {};
    virtual void inserir_elemento(int elemento) = 0;
    virtual void imprimir() = 0;
    virtual int tamanho() = 0;
};
#endif
```

Olhando para as implementações

Caso da Lista

```
#ifndef PDS2_LISTADUPLA_H
#define PDS2_LISTADUPLA_H

// . . .

class ListaDuplamenteEncadeada : public ColecaoIF {
private:
    node_t *_inicio;
    node_t *_fim;
    int _num_elementos_inseridos;
public:
    ListaDuplamenteEncadeada();

    // Comportamento comum
    virtual ~ListaDuplamenteEncadeada();
    virtual void inserir_elemento(int elemento);
    virtual void imprimir();
    virtual int tamanho();

    // Específico
    void remove_iesimo(int i);
};
#endif
```

Podemos usar via ColecaoIF/Lista

Podemos usar apenas via Lista

Conversão de Tipos

- Uma classe, ao herdar de outra, assume o tipo desta onde quer que seja necessário
- Upcasting
 - Conversão para uma classe mais genérica
- Downcasting
 - Conversão para uma classe mais específica

Conversão de Tipos

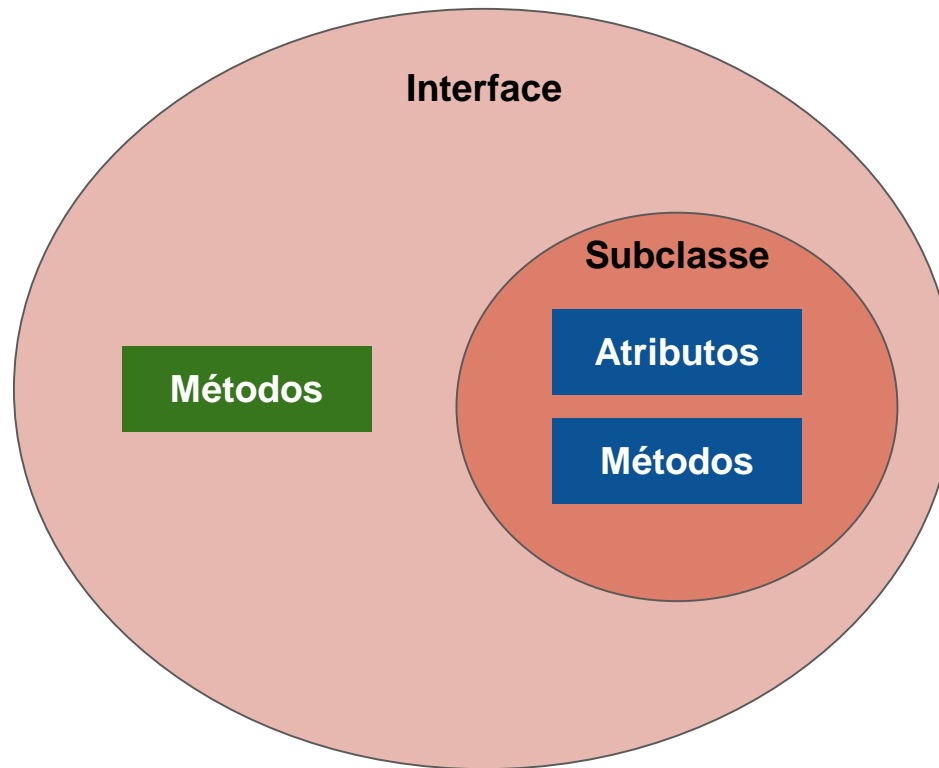
Upcasting

- ❑ Ocorre no sentido Classe \Rightarrow Interface
- ❑ Não há necessidade de indicação explícita
- ❑ A classe derivada sempre vai manter as características públicas da superclasse

Conversão de Tipos

Upcasting

Contexto de Classe



Upcasting

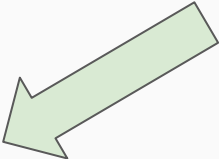
Note que os objetos viram MensagemIF auto-magicamente

```
#include "mensagem.h"
#include "mensagemimg.h"
#include "mensagemtexto.h"
#include "mensagemvoz.h"

void exibir_na_tela(MensagemIF &msg) {
    msg.exibir();
}

int main(void) {
    MensagemTexto texto("Oi, tem aula de PDS2 hoje?");
    MensagemVoz audio("audio.wav");
    MensagemImagem image("imagem03.ascii");
    MensagemTexto texto2("Mas que puxa :(");

    exibir_na_tela(texto);
    exibir_na_tela(audio);
    exibir_na_tela(image);
    exibir_na_tela(texto2);
}
```



Conversão de Tipos

Downcasting

- ❑ Ocorre no sentido Interface \Rightarrow Classe
- ❑ Não é feito de forma automática!
- ❑ Deve-se deixar explícito, informando o nome do subtipo antes do nome da variável

Conversão de Tipos

Downcasting

- ❑ Nem sempre uma superclasse poderá assumir o tipo de uma subclasse
- ❑ Toda MensagemTexto é uma MensagemIF
- ❑ Nem toda MensagemIF é MensagemTexto
- ❑ Caso não seja possível
 - ❑ Segmentation fault

Exemplo: Uno

Jogo de Uno / 8 maluco / Mau Mau

- Cada Jogador tem recebe 7 cartas
- O resto do Baralho é oculto

Jogo de Uno / 8 maluco / Mau Mau

- Cada Jogador tem recebe 7 cartas
- O resto do Baralho é oculto
- Quais as classes até agora?

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto
- Um baralho é composto de?

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto
- Um baralho é composto de?
 - Coleção de **Cartas**

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto
- Um baralho é composto de?
 - Coleção de **Cartas**
- Uno é um Jogo interessante.
 - Inicia no sentido horário, pode mudar

Jogo de Uno / 8 maluco / Mau Mau

- ☐ Cada **Jogador** tem recebe 7 **Cartas**
- ☐ O resto do **Baralho** é oculto
- ☐ Um baralho é composto de?
 - ☐ Coleção de **Cartas**
- ☐ Uno é um Jogo interessante.
 - ☐ Inicia no sentido horário, pode mudar
 - ☐ Isto é? Mantém um _____

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto
- Um baralho é composto de?
 - Coleção de **Cartas**
- Uno é um Jogo interessante.
 - Inicia no sentido horário, pode mudar
 - Isto é? Mantém um **estado**

Jogo de Uno / 8 maluco / Mau Mau

- Cada **Jogador** tem recebe 7 **Cartas**
- O resto do **Baralho** é oculto
- Um baralho é composto de?
 - Coleção de **Cartas**
- Uno é um **Jogo** interessante.
 - Inicia no sentido horário, pode mudar
 - Isto é? Mantém um **estado**
 - Nova classe, atributo **sentido**

Jogo de Uno / 8 maluco / Mau Mau

- Ao modelar o mundo real:
 - **Definir objetos**
 - **Definir responsabilidades**
 - **Definir iterações**

Cartas

- Cada carta tem uma cor e um número
- Existem cartas especiais

Cartas

- Cada carta tem uma cor e um número
- Existem cartas especiais
 - Bom local para fazer uso de?

Cartas

- ❑ Cada carta tem uma cor e um número
- ❑ Existem cartas especiais
 - ❑ Bom local para fazer uso de?
 - ❑ **Polimorfismo**
- ❑ Cartas especiais podem:
 - ❑ Alterar o sentido do jogo
 - ❑ Pular jogadores
 - ❑ Ser jogada em qualquer momento
 - ❑ Aumentar número de cartas do adversário

Jogadores

- Tem uma pontuação
- 7 cartas iniciais.
- **Porém**
 - Pode aumentar, com uma carta especial de um adversário
- **Vector/Set**

User Stories

- ☐ Iniciar Jogo
- ☐ Realizar Jogada
- ☐ Fechar programa
 - ☐ Desistir
- ☐ Salvar jogo
 - ☐ Continuar no futuro

Programação e Desenvolvimento de Software 2

Gerenciamento de Memória

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Gerenciamento de Memória

- ☐ O que provavelmente irá acontecer se tentarmos executar o código abaixo?

```
double vetor[21000000000];  
    ...  
cout << "Fim!" << endl;
```

Gerenciamento de Memória

- ☐ O que irá acontecer se tentarmos executar o código abaixo?

```
double vetor[21000000000];  
//...  
cout << "Fim!" << endl;
```

- ☐ Provavelmente “Fim” não irá aparecer
 - ☐ Estouro da pilha

Gerenciamento de Memória

☐ Já estamos bastante habituados a resolver este tipo de situação... como?

Gerenciamento de Memória

- ☐ Já estamos bastante habituados a resolver este tipo de situação... como?

```
double *vetor = new double[21000000000];  
  
//...  
  
delete vetor;  
  
cout << "Fim!" << endl;
```

- ☐ Uso de new, delete
- ☐ Se existir a quantidade necessária de memória, vai funcionar!

Qual o problema?

- ☐ O desenvolvedor C++
 - ☐ Quando ele aloca uma memória dinamicamente, por exemplo, e esquece de desalocar essa memória.
- ☐ Memory leaks
- ☐ E se pudéssemos, por exemplo, fazer new sem a necessidade de delete?
 - ☐ RAI, Smart Pointers

RAII

Aquisição de Recurso É Inicialização (RAII)

- ☐ Padrão de projeto de software para C++
- ☐ Combina a aquisição e liberação de recursos com inicialização e destruição de objetos
- ☐ Uso de memória:
 - ☐ inicia-se na declaração
 - ☐ termina quando o objeto sai do escopo (fim da execução ou lançamento de exceção)

Aquisição de Recurso É Inicialização (RAII)

- ☐ Fundamentos:

- ☐ O recurso é liberado no destruidor (por exemplo, fechando um arquivo)
- ☐ Instâncias da classe são alocadas em pilha (e não no heap)
- ☐ O recurso é adquirido no construtor (por exemplo, abrir um arquivo). Esta parte é opcional, mas comum.

Aquisição de Recurso É Inicialização (RAI)

- ☐ "Aquisição de recursos" do RAI é onde você começa algo que deve ser finalizado posteriormente, como por exemplo:
 - ☐ Abrir um arquivo (que deve ser fechado mais tarde)
 - ☐ Alocar alguma memória (e desalocá-la depois)

Aquisição de Recurso É Inicialização (RAI)

- ☐ "Inicialização" do RAI significa que a aquisição ocorre dentro do construtor
 - ☐ A abertura de um arquivo via construtor

Gerenciamento de Arquivos

```
#include <cstdio>

class Arquivo{
    std::FILE* ptr_arquivo;
public:
    Arquivo(const char* nome_arquivo)
        : ptr_arquivo(std::fopen(nome_arquivo, "w+")){
        //...
    }
    ~Arquivo(){
        //...
    }
    void escreve(const char* texto){
        //...
    }
};
```

Gerenciamento de Arquivos

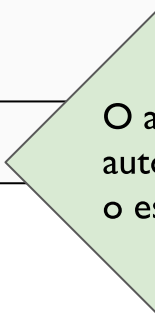
☐ O uso da classe...

```
void exemplo_uso() {  
  
    Arquivo arquivoLog("arquivoLog.txt");  
  
    Arquivo.escreve("Olá, log da aula de PDS2!");  
  
}
```

Gerenciamento de Arquivos

☐ Exemplo de uso da classe...

```
void exemplo_uso() {  
  
    Arquivo arquivoLog("arquivoLog.txt");  
  
    Arquivo.escreve("Olá, log da aula de PDS2!");  
  
}
```



O arquivo é fechado automaticamente quando o escopo termina

Gerenciamento de Arquivos

- ☐ A classe Arquivo encapsula o gerenciamento do recurso *FILE, adquirindo e liberando automaticamente a memória
 - ☐ O arquivo é fechado automaticamente quando seu escopo termina

Smart Pointers (Ponteiros Inteligentes)

Ponteiros Inteligentes

- ☐ Implementações para auxiliar a manipulação de ponteiros
- ☐ São objetos que armazenam ponteiros para objetos alocados dinamicamente (no heap)
- ☐ Seu funcionamento é similar ao de um ponteiro tradicional

Mas, qual a diferença?

- ☐ Eles deletam automaticamente o objeto apontado no momento certo (após o término da utilização)
 - ☐ Provém facilidade de desalocação automática de memória
 - ☐ Prevenção de memory leaks
- ☐ São úteis para assegurar a destruição do objeto apontado em caso de exceção

Principais Vantagens

- ☐ Não precisamos nos lembrar de liberar a memória que foi alocada para uso do objeto
- ☐ Não é necessário usar **delete** e **free** em todos os objetos que foram declarados
- ☐ Elimina o risco de dangling pointers, que ocorrem quando os ponteiros apontam para objetos já deletados
- ☐ Consequentemente, redução da ocorrência de bugs

Tipos Mais Comuns

- ☐ `unique_ptr` (antigo `auto_ptr`)
 - ☐ Ponteiro único
 - ☐ Permite um ponteiro por vez
- ☐ `shared_ptr`
 - ☐ Ponteiro compartilhado (vários proprietários)
 - ☐ Contador de referências
- ☐ `weak_ptr`
 - ☐ uso em conjunto com `shared_ptr`
 - ☐ posse temporária

Ponteiros

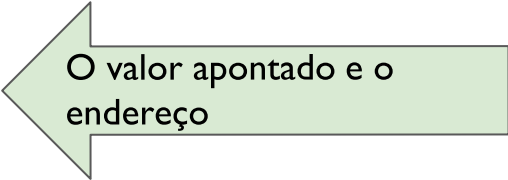
```
#include <iostream>

using namespace std;

int main() {
    int *pnum = new int();
    *pnum = 10;
    cout << *pnum << " -- " << pnum << endl;
    delete pnum;

    return 0;
}
```

```
$ ./main
10 -- 0x6e1028
```



O valor apontado e o endereço

Ponteiros Inteligentes

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    int *pnum = new int();
    *pnum = 10;
    cout << *pnum << " -- " << pnum << endl;
    delete pnum;

    return 0;
}
```



include na biblioteca
memory

unique_ptr

□ Ponteiro único

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //int *pnum = new int();
    unique_ptr<int>pnum(new int);
    *pnum = 10;
    cout << *pnum << " -- " << &pnum << endl;

    //delete pnum;

    return 0;
}
```



Definição do tipo <type>

unique_ptr

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //int *pnum = new int();
    unique_ptr<int>pnum(new int);
    *pnum = 10;
    cout << *pnum << " -- " << &pnum << endl;

    //delete pnum;

    return 0;
}
```

Incluo '&' (notação de endereço)

Não preciso mais do delete

```
$ ./main
10 -- 0x6e1028
```

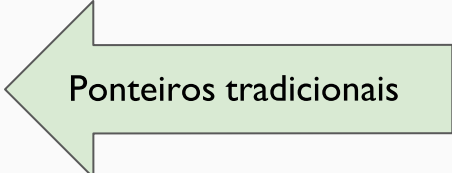

Strings

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    string *str = new string("Aula PDS2");
    cout << *str << " -- Tamanho: " << str->size() << endl;
    delete str;

    return 0;
}
```



Ponteiros tradicionais

```
$ ./main
Aula PDS2 -- Tamanho: 9
```

unique_ptr

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //string *str = new string("Aula PDS2");
    unique_ptr<string>str(new string("Aula PDS2"));
    cout << *str << " -- Tamanho: " << str->size() << endl;
    delete str;

    return 0;
}
```

```
$ ./main
Aula PDS2 -- Tamanho: 9
```

Classes

```
#include <iostream>
#include <memory>
```

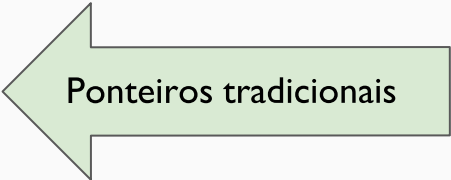
```
using namespace std;
```

```
class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    Aluno *a = new Aluno();
    cout << "Nota: " << a->getNota() << endl;
    delete a;

    return 0;
}
```

```
$ ./main
Nota: 0
```



Ponteiros tradicionais

Classes

```
#include <iostream>
#include <memory>
```

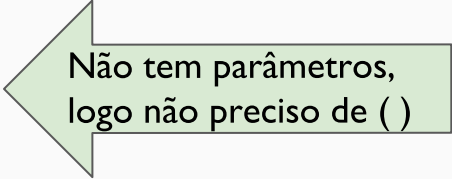
```
using namespace std;
```

```
class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```

```
$ ./main
Nota: 0
```



Não tem parâmetros,
logo não preciso de ()

Classes

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};

int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b=a;
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```



Classes

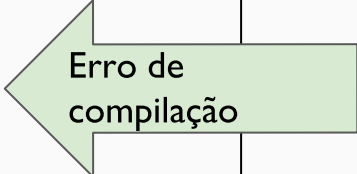
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b=a;
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```



Erro de
compilação

☐ Permite um único ponteiro por vez

shared_ptr

□ Ponteiro compartilhado

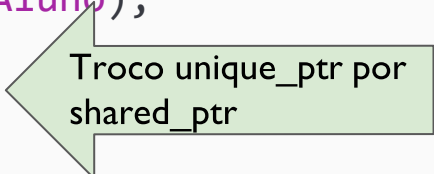
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```



Troco unique_ptr por shared_ptr

shared_ptr

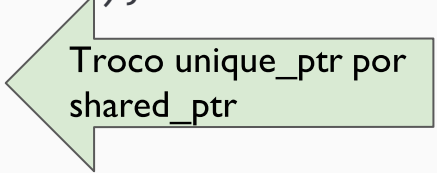
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```



Troco unique_ptr por shared_ptr

☐ O que será impresso?

shared_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 80
Nota b: 80
```

☐ Por que?

shared_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    //b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 70
Nota b: 70
```

☐ Eles estão usando o mesmo ponteiro

Qual a solução neste caso?

Qual a solução neste caso?

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b(new Aluno);
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 70
Nota b: 80
```

Utilizando listas de inicialização

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    const char* nome;
    int nota;
```

```
    Aluno(const char* n, int nt):
nome(n), nota(nt){
    //...
}
};
```

```
int main() {
    unique_ptr<Aluno>a(new Aluno("Julio",90));

    cout << "Nome: " << a->nome << endl;
    cout << "Nota: " << a->nota << endl;

    return 0;
}
```

```
$ ./main
Nome: Julio
Nota: 90
```

Programação e Desenvolvimento de Software 2

Tratamento de exceções

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

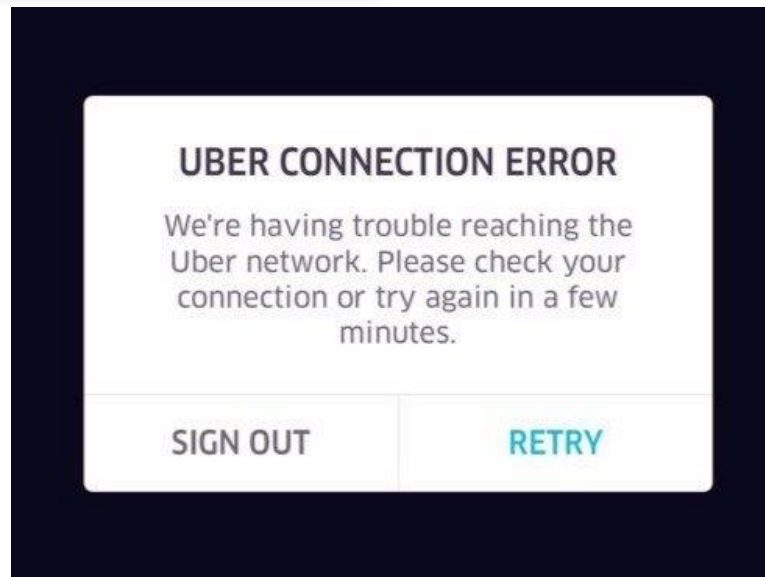
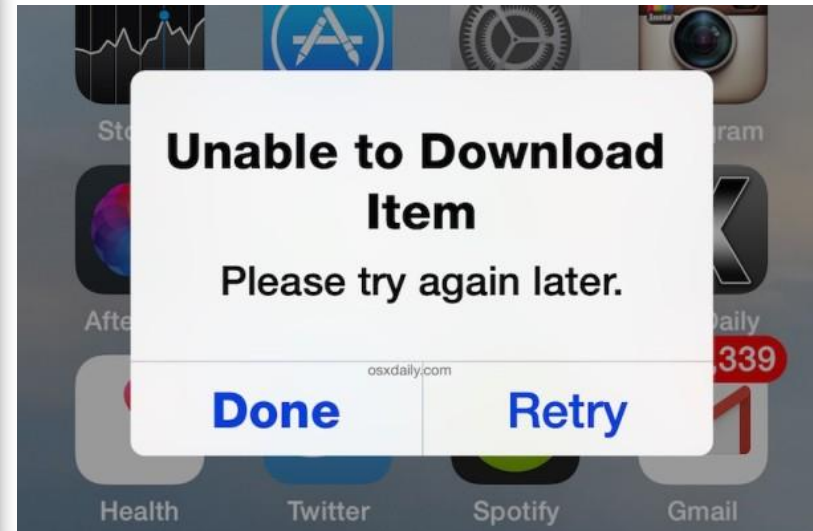
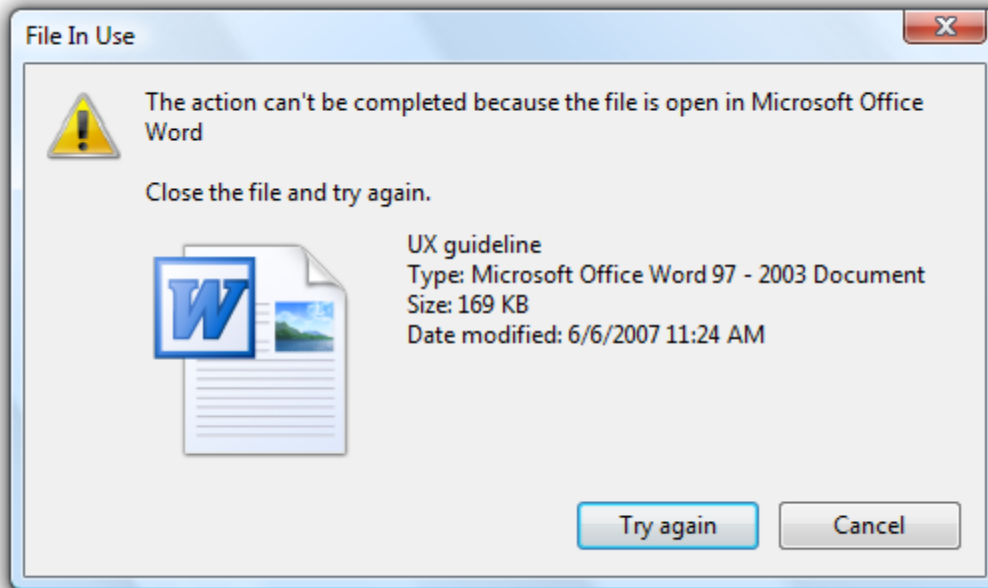
Introdução

- ☐ O que é uma exceção?
 - ☐ “Exceptional event”
 - ☐ Algum evento/acontecimento ‘inesperado’ que ocorre no contexto da execução do programa
- ☐ Importante tratar e gerenciar esses eventos
 - ☐ Diferente das asserções (erros fatais)
 - ☐ Problema que pode não ser somente do código
 - ☐ Demandam alteração no fluxo de execução

Introdução

- ☐ O que pode gerar uma exceção?
 - ☐ Entradas inválidas, falhas de hardware, ...
 - ☐ Exemplos:
 - ☐ Timeout ao enviar dados pela rede
 - ☐ Erros na leitura de arquivos
 - abrir um arquivo inexistente
 - ☐ Tentativas de acessos inválidos
 - uma posição inválida em um vetor

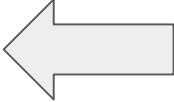
Exceções



Introdução

- Como sabemos, a grande maioria dos erros não podem ser detectados em tempo de compilação
- Alguns erros são bugs no programa
- O que fazer então nesses casos?

Tratamentos

- ☐ Definir valor de uma variável global
- ☐ Convenção de códigos de retorno
 - ☐ C/C++: 0 e !0
 - ☐ Java: boolean
- ☐ Retornar a mesma resposta da vez anterior
 - ☐ Retornar valor válido mais próximo
- ☐ Chamar rotina de processamento de erros
- ☐ Lançar uma exceção e tratá-la! 

Motivação

- ❑ Gerar programas mais robustos
- ❑ Permitem ao código/usuário agir
 - ❑ Re-conectar
 - ❑ Escolher outro arquivo
 - ❑ Outro parâmetro
- ❑ Simples de usar
- ❑ Alguém tem que **tratar** a exceção

Exceções

- Maneira facilitada de informar que a rotina não deve (pode) continuar a execução
- Sinalização da existência de um erro
 - É criada uma variável que representa a falha
 - A exceção deve então ser “lançada”
 - O código é desviado da execução normal
- Tratamento
 - A “captura” da exceção também deve ser feita

Exceções

□ C++

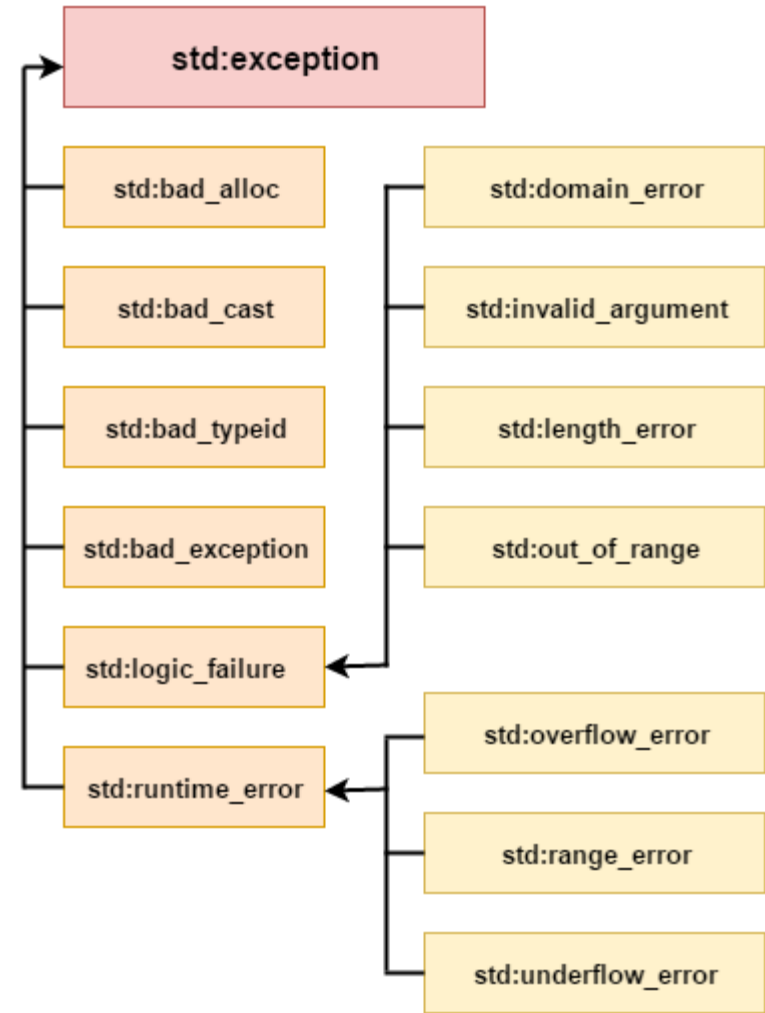
- Tratamento estruturado (parte da linguagem)
- Mais poderoso e flexível que códigos de retorno
- `try-throw-catch`

□ Definidas como classes (ou qualquer tipo)

- Vantagens do paradigma de OO
- Contém informações sobre o erro (contexto)
- Pré-definidas / Criadas pelo programador

Tipos Comuns de Exceções

- C++ já tem exceções comuns na biblioteca padrão
- Podemos definir novos tipos para erros específicos do programa



Exemplo

Qual o problema com o código abaixo?

```
#include <string>
#include <iostream>

int main() {
    std::string texto;
    std::cin >> texto;
    texto.substr(10);
    return 0;
}
```


Exemplo

- Ao executar o código com uma entrada com menos do que 10 caracteres:

**libc++abi.dylib: terminating
with uncaught exception of type
std::out_of_range: basic_string
Abort trap: 6**

Tratando Exceções

- ❑ Exceções podem ser tratadas
- ❑ Ou lançadas para frente
- ❑ Para tratar: fazemos uso de **try/catch**
- ❑ Para lançar: fazemos uso de **throw**
 - ❑ Existem casos onde uma função/método não sabe tratar um erro. Repassa o mesmo
 - ❑ Em algum momento chegamos no main

Exceções

- Observada pela instrução **try**
 - Região protegida (observável)
 - Bloco de código onde pode ocorrer a exceção
- Capturada pela instrução **catch**
 - Bloco específico para cada tipo de exceção
 - Responsável pelo tratamento (manipulação)

Exemplo com Métodos

```
#include <string>
#include <iostream>

std::string pega_sub_string(std::string str, int k) {
    return str.substr(k);
}

std::string le_entrada() {
    std::string texto;
    std::cin >> texto;
    return pega_sub_string(texto, 10);
}

int main() {
    std::cout << le_entrada();
    return 0;
}
```

Tratando Exceções

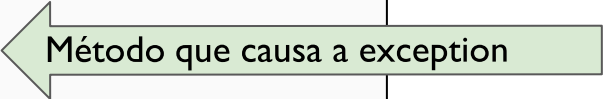
Usamos o **try/catch**

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```

Tratando Exceções

Usamos o **try/catch**

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```



Método que causa a exception

Tratando Exceções

Neste caso, é um bom tratamento?

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```

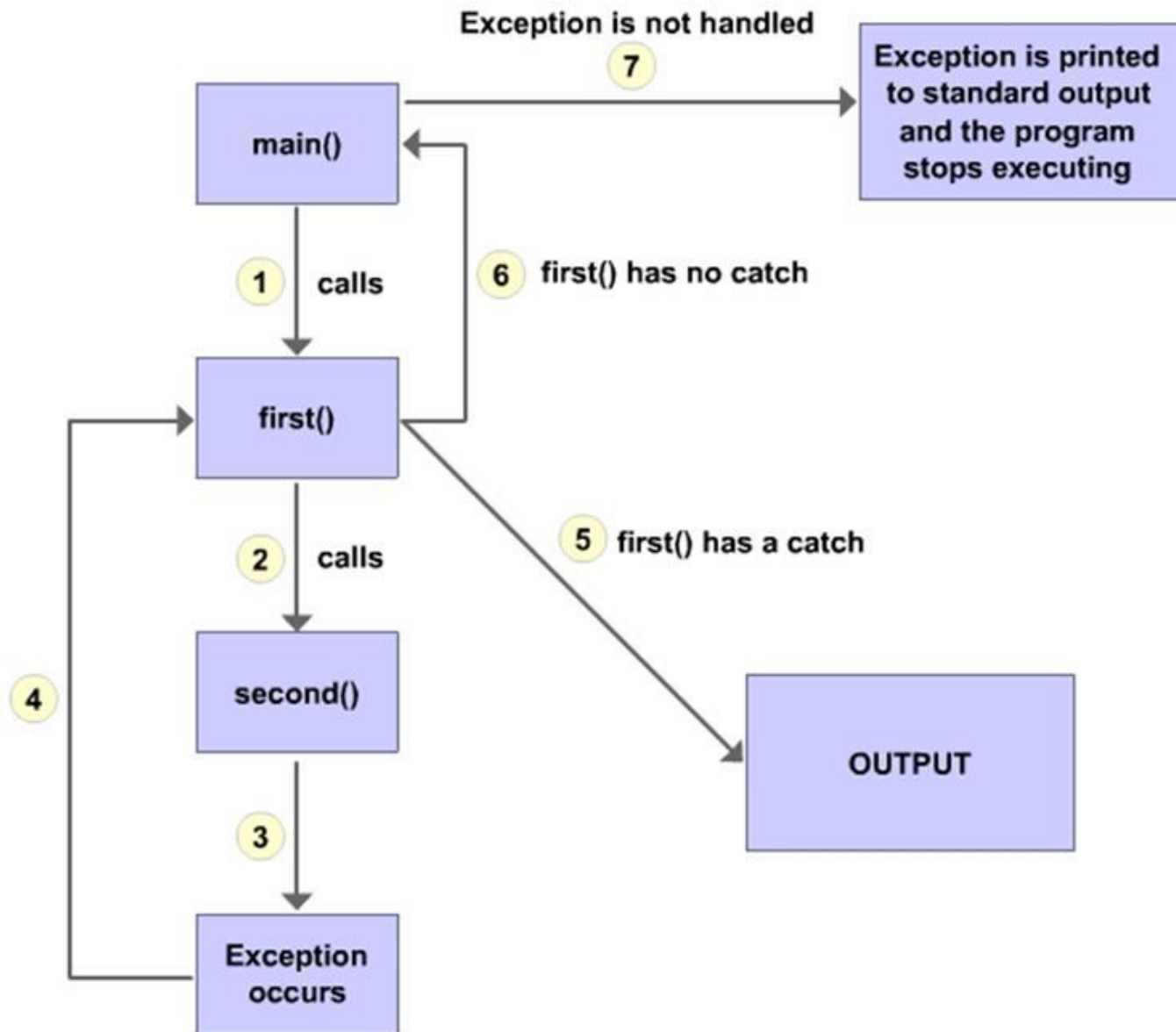
← Método que causa a exception

← Tratamento

Tratando Exceções

- ❑ Idealmente teremos uma ação a ser seguida. Caso contrário, é melhor repassar o erro para frente
- ❑ Ao não realizar o catch, a exceção continua sendo lançada na pilha de chamadas

Stack Unwind



Tratamento Melhor

- Temos uma ação
- Continuar no laço até a entrada ser ok!

```
std::string le_entrada() {  
    std::string texto;  
    while (1) {  
        try {  
            std::cin >> texto;  
            return pega_sub_string(texto, 10);  
        } catch (std::out_of_range &e) {  
            std::cerr << "Entrada invalida! Digite novamente.\n";  
        }  
    }  
}
```

Lançando Exceções

- Existem situações que nosso código deve lançar uma exceção
- Operador **throw**
 - Sempre dentro de um bloco
 - Se nada tratar (catch), o programa terminará
- A exceção lançada é um objeto
 - Previamente instanciado
 - Instanciado no momento do lançamento
 - Tipo deve ser parâmetro de um bloco `catch`

Lançando

- Existem situações que nosso código deve lançar uma exceção. Usamos **throw**

```
#include <stdexcept>

int fatorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Não existe fatorial de n < 0");
    }
    if (n <= 1) {
        return 1;
    }
    return n * fatorial(n-1);
}
```

Lançando

- ❑ Escolha uma exceção de acordo com o erro. Podemos lançar mais de uma
- ❑ Por exemplo, a maioria dos computadores não vai computar o fatorial de $n \geq 20$ corretamente.
 - ❑ Overflow: -2102132736
- ❑ Como sinalizar para o usuário?

Lançando duas Exceções

- ❑ Escolher a exceção correta para o caso
- ❑ Precisamos tratar e testar as duas

```
int fatorial(int n) {  
    if (n < 0) {  
        throw std::invalid_argument("Não existe fatorial de n < 0");  
    }  
    if (n >= 20) {  
        throw std::overflow_error("Não consigo computar para n>=20");  
    }  
    if (n <= 1) {  
        return 1;  
    }  
    return n * fatorial(n-1);  
}
```

Exemplo Multicatch

- ❑ e.what() imprime o erro
- ❑ Qual o problema do código abaixo?

```
int main() {  
    try {  
        std::cout << fatorial(-2);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    }  
}
```

Exemplo Multicatch

- ❑ e.what() imprime o erro
- ❑ Qual o problema do código abaixo?
 - ❑ Não tratamos o caso a seguir

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    }  
}
```


Exemplo Multicatch

- ❑ e.what() imprime o erro
- ❑ Qual o problema do código abaixo?
 - ❑ Resolvendo

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    } catch (std::overflow_error &e) {  
        std::cout << e.what();  
    }  
}
```

Exemplo Multicatch

- ❑ e.what() imprime o erro
- ❑ Qual o problema do código abaixo?
 - ❑ Qual o problema agora?

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    } catch (std::overflow_error &e) {  
        std::cout << e.what();  
    }  
}
```

Hierarquia de Exceções

- A definição de qual bloco **catch** vai ser executado depende de dois fatores:
 - 1) Tipo
 - 2) Ordem
- Assim:
 - Logo que o tipo casar com um dos blocos **catch** vamos entrar no bloco
 - Podemos explorar herança

Pegando Exceções Genéricas

- Agora o código funciona com a exceção genérica: **exception**

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::exception &e) {  
        std::cout << e.what();  
    }  
}
```

Ajudando o usuário do método

- Podemos usar **noexcept** para
 - Definir que uma função nunca lança
- Ou podemos usar **noexcept(false)**
 - Deixando claro que a função pode lançar
- Ou **throw**
 - Indica o tipo que pode ser lançado

```
void f() noexcept; // the function f() does not throw
void f() noexcept(false); // g may throw
void f() throw(std::invalid_argument); // lança aquele tipo
```

Definindo Exceções

- ❑ Em C++ podemos lançar qualquer coisa para frente
- ❑ Idealmente, lançaremos uma sub-classe da classe **std::exception**
 - ❑ Deixando claro que é um erro
- ❑ Porém podemos fazer:
 - ❑ **throw "ocorreu um erro";**

Definindo Exceções

- ❑ Sugiro usar herança na classe exception
- ❑ Lembrando, podemos fazer catch ou na super-classe ou na sub-classe
- ❑ Dois exemplos

```
class ContaSemSaldoException : public std::exception {  
    // . . . codigo aqui  
};
```

```
class ContaSemSaldoException : public std::invalid_argument {  
    // . . . codigo aqui  
};
```

Definindo Exceções

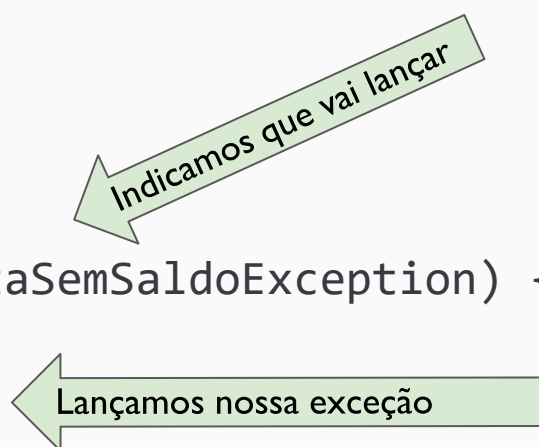
- ❑ Podemos sobrescrever os métodos da classe base. São **virtual**
- ❑ No exemplo abaixo definimos o nosso **what**, podemos usar qualquer mensagem

```
class ContaSemSaldoException : public std::exception {  
public:  
    virtual const char* what() const noexcept override;  
};
```

```
const char* ContaSemSaldoException::what() const noexcept {  
    return "Conta sem saldo!";  
}
```


Uso da Exceção

```
class Conta {  
private:  
    int _agencia;  
    int _numero;  
    double _saldo = 0;  
    bool possui_saldo(double valor) {  
        return (_saldo - valor) > 0;  
    }  
public:  
    void sacar(double valor) throw(ContaSemSaldoException) {  
        if (!possui_saldo(valor)) {  
            throw ContaSemSaldoException();  
        }  
        this->_saldo -= valor;  
    }  
};
```



Indicamos que vai lançar

Lançamos nossa exceção

Programação e Desenvolvimento de Software 2

Testes, geração de casos de teste e teste de unidade

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Introdução

- Modificar um programa é difícil
 - Mais do que implementá-lo inicialmente
 - Modificações em cadeia no código
 - Correções introduzem (novos) erros
- Como diminuir a chance de erros futuros?
 - Testar o código durante desenvolvimento
 - O que é um erro no programa? E um teste?

Introdução

- O que é teste de software?
 - Atividade responsável por avaliar as capacidades de um programa, verificando o alcance de resultados previamente estabelecidos

“Testing is the process of executing a program with the intent of finding errors.”

- Glenford. F. Myers, The Art of Software Testing, p. 6

Introdução

Motivação

- Diminuir o número de erros ao cliente
 - Melhorar a qualidade do software
- Detectar problemas mais rapidamente e de forma antecipada
 - Minimizar o custo de correção
- Modelagem mais precisa
 - Pensar em possíveis testes (cenários) para o sistema ajudam a entender melhor o problema

Introdução

Motivação

Table 7-5. Hours to fix bug based on introduction point

| | | STAGE FOUND | | | |
|---------------------|--------------|---------------------|-------------|--------------|----------------------|
| Stage Introduced | Requirements | Coding/Unit Testing | Integration | Beta Testing | Post-product Release |
| Requirements | 1.2 | 8.8 | 14.8 | 15.0 | 18.7 |
| Coding/Unit testing | NA | 3.2 | 9.7 | 12.2 | 14.8 |
| Integration | NA | NA | 6.7 | 12.0 | 17.3 |

NA = Not applicable because cannot find a bug before it is introduced

<https://blog.fullstory.com/what-we-learned-from-google-code-reviews-arent-just-for-catching-bugs/>

Introdução

Princípios

- Teste \neq Debugging

- Caso o teste encontre um erro, o processo de depuração pode ser usado para corrigi-lo

- Programa \rightarrow Paciente Doente

- Teste de sucesso \rightarrow Problemas detectados

- Teste sem sucesso \rightarrow Nenhum problema

- Ponto de vista psicológico

- Análise e Codificação são tarefas construtivas

- Teste é uma tarefa “destrutiva”

Tipos de Testes

- ☐ Testes de unidade
 - ☐ Programação/codificação (módulo específico)
 - ☐ Nível de classe
- ☐ Testes de integração
 - ☐ Projeto (diferentes módulos)
- ☐ Testes de validação
 - ☐ Requisitos
- ☐ Testes de sistemas
 - ☐ Demais Elementos

Tipos de Testes

- Outros tipos de testes
 - Instalação
 - Segurança
 - Regressão
 - Performance
 - Usabilidade
 - [...]
 - Black-box vs White-box

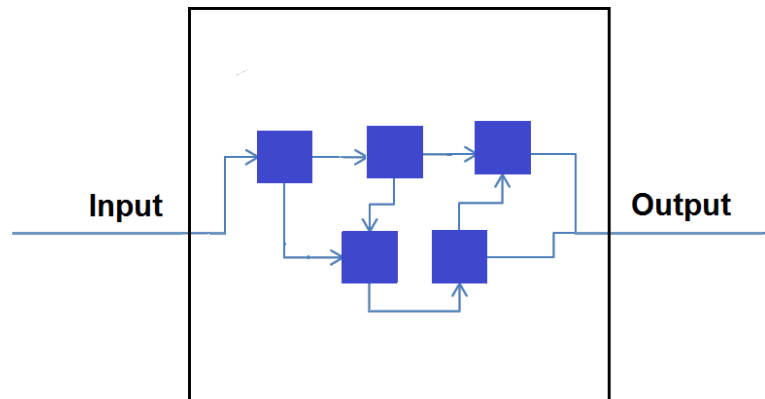
Black-box v. White-box

Black-box. Pouco, ou zero, acesso ao código. Por exemplo, podemos testar um sistema já pronto com usuários.



White-box testing

- Conhecemos o código e como o mesmo funciona. Podemos encaixar e manipular vários módulos do mesmo



Testes de unidade

- Testes de Unidade (White-box)
 - Nosso foco
 - Código feito para testar as classes
- Parece cíclico
 - Não é. Sabendo do contrato sabemos como usar os objetos de módulos
 - O teste é um código cliente
 - Pouca ou quase zero lógica

Teste de Unidade

- Trecho de código que chama outro trecho de código para verificar o comportamento apropriado de uma determinada hipótese
- Hipótese não validada (resultado incorreto), dizemos que o teste de unidade falhou
 - O objetivo é que todos os testes passem!
 - Resultados de acordo com o esperado

Testes de Unidade

O que é uma unidade?

- Menor unidade de classe testável
 - Método/bloco de código de um método
- Teste verifica uma hipótese para o método
 - As partes que utilizam o método devem ser testadas em outros casos de testes separados
- Diferentes aspectos podem ser testados
 - E/S, condições de contorno, exceções, ...

Testes de Unidade

Casos de teste

- ☐ Condição particular a ser testada
 - ☐ Valores de entrada
 - ☐ Restrições de execução
 - ☐ Resultado ou comportamento esperado

IEEE Standard 610 (1990) defines test case as follows:

1. A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
2. (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

Testes de Unidade

Casos de teste

- ☐ Refletem os requisitos que serão verificados
 - ☐ Casos Básicos
 - ☐ Positivo
 - ☐ Demonstrar que o requisito é atendido
 - ☐ Negativo
 - ☐ Requisito só é atendido sob certas condições
 - ☐ O que acontece em cenários com condições especiais ou dados inaceitáveis, anormais ou inesperados?

Testes de Unidade

Exemplo I

- ☐ Programa para identificar triângulos
 - ☐ Entrada: 3 números inteiros (lados)
 - ☐ Saída: Equilátero, Isósceles, Escaleno
- ☐ Casos Positivos
 - ☐ Quantos casos de teste para equilátero?
 - ☐ [5,5,5]
 - ☐ Quantos casos de teste para isósceles?
 - ☐ [3,3,4]; [3,4,3]; [4,3,3]
 - ☐ Quantos casos de teste para escaleno?
 - ☐ [3,4,6]; [3,6,4]; [4,3,6]

Testes de Unidade

Exemplo I

☐ Casos Negativos

☐ Teste quando um dos lados é zero

☐ Teste quando um dos lados é negativo

☐ Teste verificando valores para triângulos válidos

☐ Verificar diferentes permutações

☐ [1,2,3]; [1,3,2]; [2,1,3]; [2,3,1]; [3,1,2];
[3,2,1]

Testes de Unidade

Vantagens

- ☐ Permitem a utilização de ferramentas que validam o código por condições fail/pass
- ☐ Podem ser feitos pelo implementador
- ☐ Ajudam a entender e manter o código
- ☐ Falhas detectadas durante as alterações
 - ☐ Se o código muda, o teste começa a falhar

Framework

- A automatização dos testes de unidade
 - Agilizar a verificação após mudanças
 - Evitar um trabalho tedioso (caro) → Falhas
- Doctest: <https://github.com/onqtam/doctest>
 - Estamos usando nas VPLs
 - Light, fast, single-header, free, feature-rich, ...
- Outras opções:
 - Catch2: <https://github.com/catchorg/Catch2>
 - GoogleTest: <https://github.com/google/googletest>

Framework

- ❑ Funcionamento baseado em asserções
- ❑ Diferentes níveis de severidade
 - ❑ REQUIRE / CHECK / WARNING
- ❑ Métodos auxiliares
 - ❑ Condições
 - ❑ `CHECK(thisReturnsTrue()) ;`
 - ❑ Exceções
 - ❑ `CHECK_THROWS_AS(func(), std::exception) ;`

Framework - Macros doctest

- Existe uma série de macros no doctest
- A maioria é descrita aqui
- <https://github.com/onqtam/doctest/blob/master/doc/markdown/assertions.md>

Framework

- ❑ Criar um arquivo teste (!)
 - ❑ Geralmente um para cada classe
- ❑ Criar um método de teste (test case)
 - ❑ Criar um cenário de teste
 - ❑ Executar a operação sendo testada
 - ❑ Conferir o resultado retornado

Exemplo Simples

- Vamos fazer um código que gera um fatorial. Estilo o primeiro VPL

```
#ifndef PDS2_FAT_H  
#define PDS2_FAT_H  
  
int fatorial(int);  
  
#endif
```


Exemplo Simples

- ❑ O código está incompleto
- ❑ Apenas para compilar

```
#include "fatorial.h"

int fatorial(int n) {
    return n;
}
```

○ Teste

- Abaixo temos um teste simples
- Vamos entender o mesmo

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(1) == 1);
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(10) == 3628800);
}
```

Entendendo o nosso código

- Um teste de unidade é representado por um código C++
 - Temos algumas macros definidas pela biblioteca doctest (outras funcionam de forma similar)
- Por isso iniciamos com o include

Entendendo o nosso código

```
#include "doctest.h"  
#include "fatorial.h"
```

← Include doctest e código além do código que será testado

```
TEST_CASE("Testando o fatorial") {  
    CHECK(fatorial(2) == 2);  
    CHECK(fatorial(3) == 6);  
    CHECK(fatorial(4) == 24);  
    CHECK(fatorial(10) == 3628800);  
}
```

Test Cases

- Cada Test Case foca em uma funcionalidade. Geralmente método

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}
```



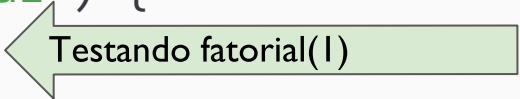
Caso de teste

Test Cases

- A macro CHECK verifica se o resultado é igual ao esperado

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}
```



Executando

- ❑ Vamos usar um main a moda antiga
- ❑ Compilar tudo e rodar

```
$ g++ -std=c++14 *.cpp -o main  
$ ./main
```

□ Parece que deu erro

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE:  Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```


□ Alguns testes passam

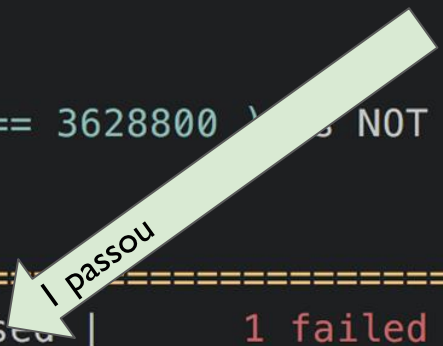
```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```



❑ Outros testes falham.

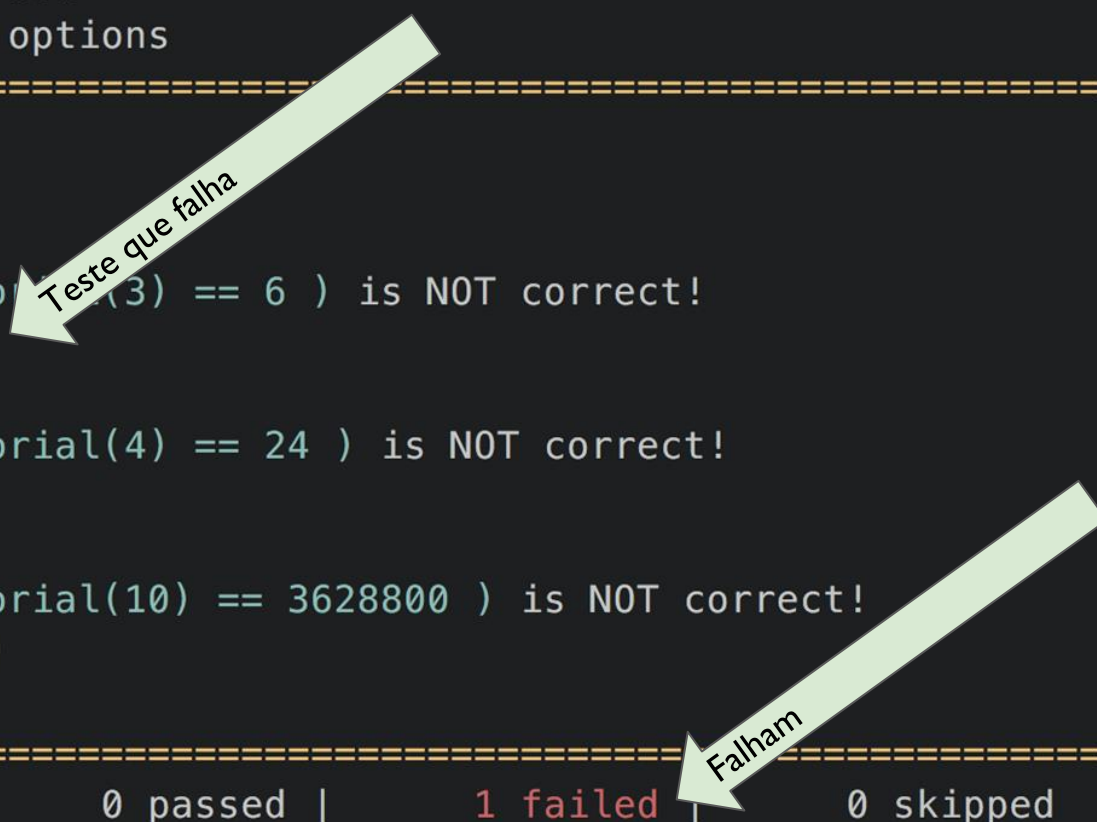
```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```



Saída completa

```
$ ./main -s
```

□ Use a opção -s

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options

=====

testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:5: SUCCESS: CHECK( fatorial(2) == 2 ) is correct!
  values: CHECK( 2 == 2 )

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====

[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```

Vamos corrigir o programa

- Nova versão do código
- Parece ok?
 - Ainda temos erro. fatorial(0);

```
#include "fatorial.h"

int fatorial(int n) {
    if (n <= 1) return n;
    return n * fatorial(n-1);
}
```

Rodando testes novamente

- Parece que tudo está executando corretamente
- Qual foi o problema?

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
[doctest] test cases:      1 |      1 passed |      0 failed |      0 skipped
[doctest] assertions:     4 |      4 passed |      0 failed |
[doctest] Status: SUCCESS!
```

Nosso Teste

- ❑ Nunca vai testar fatorial(0)
- ❑ Deixa o código com erro

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}
```

Teste Completo

- Dois casos. Um para o zero
 - Especial
- Outro geral. Valores comuns do fatorial

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("1: fatorial of 0 is 1 (corner case)") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("2: factorials of 1 and higher are computed (caso geral)") {
    CHECK(fatorial(1) == 1);
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(10) == 3628800);
}
```

Valores inválidos

- Como que o código se comportaria com o fatorial(-20)?
- Ainda tem erro. Precisamos sinalizar que um valor inválido foi passado

Exceções

- ❑ Frequentemente uma função não consegue realizar a operação com uma dada entrada
- ❑ Ou o estado de um objeto é inválido
- ❑ Para sinalizar tal problema fazemos uso de exceções
- ❑ Interrompem o código
- ❑ Vimos este assunto na nossa última aula

Código Final

```
#include <stdexcept>

#include "fatorial.h"

int fatorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Não existe fatorial de n < 0");
    }
    if (n <= 1) {
        return 1;
    }
    return n * fatorial(n-1);
}
```

Teste Final

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o caso especial") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("Testando o fatorial geral") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}

TEST_CASE("Testando o caso invalido") {
    CHECK_THROWS(fatorial(-1));
}
```


Teste Final

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o caso especial") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("Testando o fatorial geral") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}

TEST_CASE("Testando o caso invalido") {
    CHECK_THROWS(fatorial(-1));
}
```



Verifica que lança uma exceção

Problemas ao usar doctest

- ❑ Temos dois mains, um do doctest
- ❑ Um main do código principal
- ❑ O g++ não deixa compilar tal caso
- ❑ Gerenciar testes, coberturas, gdb etc etc
- ❑ Como resolver?

Problemas ao usar doctest

- ❑ Temos dois mains, um do doctest
- ❑ Um main do código principal
- ❑ O g++ não deixa compilar tal caso
- ❑ Gerenciar testes, coberturas, gdb etc etc
- ❑ Como resolver?
 - ❑ Fazer um makefile que cuida de cada caso
 - ❑ Ou utilizar um já pronto

Usando o DocTest em projetos maiores

- Lembrando da nossa estrutura de projeto
- Adicionamos pastas para os testes

| | |
|--------------------------|--|
| . raiz do projeto | |
| ---- Makefile | |
| ---- build/ | [diretório] |
| ---- | [objetos compilados] |
| ---- third_party/ | [bibliotecas de outras pessoas] |
| ---- doctest.h | |
| ---- include/ | [diretório] |
| ---- src/ | [diretório com código] |
| ---- classe1.cpp | |
| ---- tests/ | [diretório com código] |
| ---- test_classe1.cpp | |

Cobertura de código

- Medida do grau que o código do programa é executado dado um conjunto de testes
 - Percentual do código que foi testado
- Quanto maior a cobertura, menor a chance do código conter erros não detectados

Cobertura de código

- Declaração
 - Testes que avaliam todas as linhas do código
 - Testes simples, porém pobres
- Decisões (branches)
 - Avaliar diferentes caminhos condicionais
- Condições
 - Parada, valores inválidos, valores limite, ...

Cobertura de código

Ferramentas

☐ gcov

- ☐ é a ferramenta para C++ que verifica a cobertura
- ☐ analisa o número de vezes que cada linha de um programa é executada durante uma execução
- ☐ permite encontrar áreas do código que não são utilizadas ou que não são avaliadas nos testes

☐ LCOV

- ☐ Formata relatórios em arquivos .html
- ☐ Facilita identificar e verificar problemas

Cobertura de código

Ferramentas - Passo-a-Passo

1. Compilar todos os arquivos com o parâmetro “--coverage” (saída arquivos ‘.gcno’).

```
g++ -c --coverage factorial.cpp
```

```
g++ --coverage -o TesteFactorial TesteFactorial.cpp factorial.o
```

2. Execute o arquivo executável (saída arquivos ‘.gcda’).

```
./TesteFactorial
```

3. Gerar os relatórios de cobertura (saída arquivos ‘.gcov’).

```
mkdir coverage
```

```
mv *.gcno *.gcda coverage/
```

```
gcov -lpr *.cpp -o coverage/
```

```
mv *.gcov coverage/
```

4. Gerar o relatório em html*.

```
lcov --no-external --capture --directory . --output-file
```

```
coverage/coverage.info
```

```
genhtml coverage/coverage.info --output-directory coverage
```

**Importante: Parece que o LCOV não é compatível com GCC 8.0.*

Exemplo Cobertura

- ❑ A linha abaixo indica que executamos todas as linhas do arquivo.
- ❑ Ou seja, cobrimos todos os casos!

```
$ g++ --coverage -std=c++14 *.cpp -o main  
$ ./main  
$ gcov fatorial.cpp  
File 'fatorial.cpp'  
Lines executed:100.00% of 6  
fatorial.cpp:creating 'fatorial.cpp.gcov'
```

Exemplo Cobertura

- ❑ Se apagarmos o último teste
 - ❑ Aquele do valor negativo
- ❑ Não cobrimos mais tudo

```
File 'fatorial.cpp'  
Lines executed:83.33% of 6  
fatorial.cpp:creating 'fatorial.cpp.gcov'
```

Cobertura de Testes

- A Cobertura é um bom sinal que cobrimos todo o código
 - Ou boa parte do mesmo
- Porém
 - Não fala nada da qualidade dos testes
 - Podemos ter 100% de cobertura com erros de lógica ainda (exemplo Fatorial)

Escrevendo bons testes

- Bons testes cobrem a maioria dos (ou todos os) fluxos possíveis de execução
- Teste diferentes formas de escrever a mesma função (overloading)
- Além de diferentes implementações de uma mesma interface
 - Um teste por classe, no mínimo

Dicas Para Escrever Testes

- Teste caso base
 - Onde seu algoritmo com certeza funciona
- Teste os *corner cases*
 - Entradas especiais
 - Ordenar um vetor com 1 elemento
- Teste os valores inválidos

Metodologias de Desenvolvimento

- ❑ Quando devemos iniciar a fase de testes?
 - ❑ Depois de terminada a codificação?
 - ❑ Não é uma boa ideia! Por que?
- ❑ Desenvolvimento Orientado por Testes (TDD)
 - ❑ Foco deve ser no requisito, não no código!
 - ❑ Interface → Comportamento
 - ❑ Vamos ver com mais detalhes nas próximas aulas...

Exercício

- Como testar peças de xadrez? Isto é, temos uma interface `Peca` com o método `boolean podeMover(int x, int y);`
- Diversas peças implementa a mesma.
- Uma classe `Tabuleiro` com um método `void move(Peca &peca, int x, int y);`

Exercício

- Precisamos de testes para cada classe que implementa a interface Peca
- Além de um teste para Tabuleiro
 - O mesmo que conhece o tamanho

Programação e Desenvolvimento de Software 2

Refatoração

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Projeto de Software

- Projeto (design de Software)
 - Requisitos do usuário -> Software
 - Estrutura do software (módulos, classes, ...)
 - O próprio código é o design!
- Mas o desenvolvimento de um software é algo dinâmico...

Evolução / Manutenção de Software

- ❑ Um software precisa evoluir
- ❑ Com a evolução
 - ❑ O código vai sendo atualizado
 - ❑ Decisões passadas vem perdendo efeito
 - ❑ Elementos inúteis sem benefícios diretos
 - ❑ Difícil fazer alterações e manter o design inicial
- ❑ O que fazer?

Refatoração

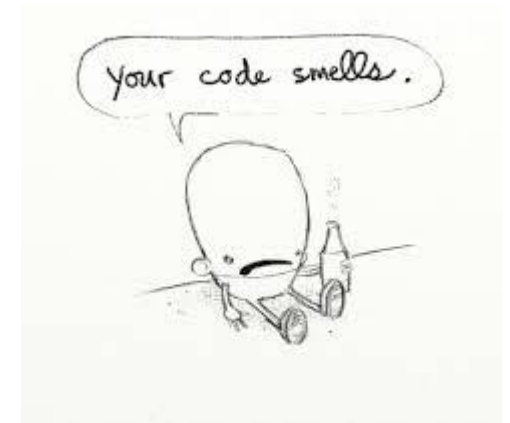
- ❑ Processo de reescrever códigos do sistema para melhorar sua estrutura de maneira geral
- ❑ Objetivo
 - ❑ Melhorar o código
 - ❑ Sem mudar as funcionalidade
 - ❑ Sem inserir bugs

Refatoração

- Modificação [pequena] no sistema que não altera o comportamento funcional, mas que melhora qualidades não funcionais
 - Flexibilidade, clareza, robustez, ...
- Alteração no design de uma aplicação
 - Atividade que estava implícita
 - Preceito básico de eXtreme Programming (XP)

Vantagens

- ❑ Melhorar aspectos como
 - ❑ Modularização
 - ❑ Reuso
 - ❑ Complexibilidade
 - ❑ Manutenabilidade
- ❑ Como?
 - ❑ Atacar os “bad smells” no código
 - ❑ Por “sorte”, temos uma série destes



Refatoração

- Design Smells/Bad Smells
 - Características (odores) que são perceptíveis em softwares (códigos) de má qualidade (podres)
 - Rigidez, Fragilidade, Imobilidade, Viscosidade, Complexidade, Repetição, Opacidade
- Propor as refatorações adequadas a partir da identificação de um desses problemas

Refatoração

Exemplos

- ❑ Mudança em nomes de variáveis e métodos
- ❑ Redução de código duplicado
 - ❑ É mais fácil fazer um “Copy and Paste”
- ❑ Generalizar/flexibilizar métodos
- ❑ Membros não encapsulados (públicos)
- ❑ Mudanças arquiteturais
 - ❑ Módulos, Classes, Interfaces, ...

Refatoração

- Não é uma reestruturação arbitrária
 - Código ainda deve funcionar (não inserir bugs)
 - Testes tentam garantir isso
 - Mudanças pequenas/pontuais (não reescrever tudo)
 - A semântica deve ser preservada
- Resultado
 - Alta coesão / Baixo acoplamento
 - Reusabilidade, legibilidade, testabilidade

Refatoração

□ Coesão

- Grau de dependência entre os elementos internos de um mesmo módulo
- Funções, responsabilidades (mesmo objetivo)

□ Acoplamento

- Grau de interdependência entre módulos
- Alteração de um demanda alteração no módulo

Refatoração

- Esses são exemplos de refatoração?
 - Adicionar novas funcionalidades
 - Melhorias no desempenho
 - Correção de erros existentes
 - Detecção de falhas de segurança

Refatoração

- Esses são exemplos de refatoração?
 - Adicionar novas funcionalidades
 - Melhorias no desempenho
 - Correção de erros existentes
 - Detecção de falhas de segurança

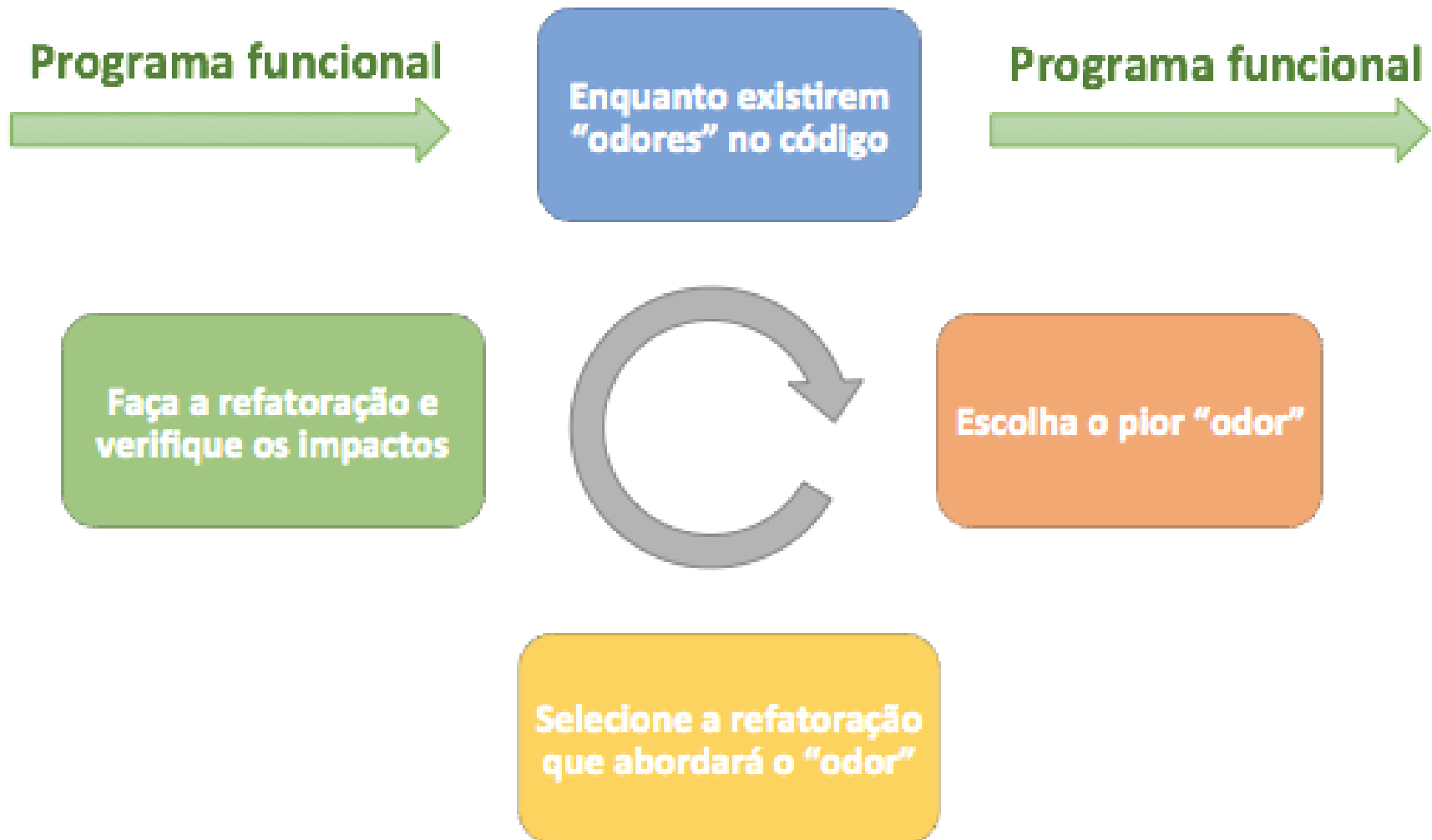
WRONG!

Refatoração

- ❑ Então, quando fazer?
 - ❑ Encontrou um “bad smell”
 - ❑ Sabe uma maneira melhor de fazer as coisas
 - ❑ Alteração não vai quebrar o código
- ❑ E quando NÃO fazer?
 - ❑ Código estável que não precisa mudar
 - ❑ Prazo para entrega se aproximando
 - ❑ Pouco conhecimento do código (terceiros)

Refatoração

Ciclo



Refatoração

- Geralmente são mudanças simples
 - Operações sistemáticas e óbvias
 - Catálogo de refatorações [Fowler, 1999]
- Localmente pode não ser tão perceptível, porém no todo o impacto é considerável

“If you want to refactor, the essential precondition is having solid tests.”

- Martin Fowler, Refactoring, p. 89

<https://refactoring.com/catalog/>

Alguns Odores Comuns

| | |
|---------------------------|---|
| Código Duplicado | <ul style="list-style-type: none">- Criar método comum- Ou criar classe- Substituir código por chamada |
| Método Longo | <ul style="list-style-type: none">- Criar sub-métodos- Ou criar classe- Inserir chamadas |
| Classe Longa | <ul style="list-style-type: none">- Criar novas classes- Extrair super-classe ou interface- Re-adequar o código |
| Inveja de Features | <ul style="list-style-type: none">- Extrair método- Mover método- Composição |
| Muita Intimidade | <ul style="list-style-type: none">- Re-organizar dados- Mover métodos- Mover campos |

Catálogo do Problemas e Soluções

- Existem diversos outros problemas
- Além de outras soluções

<https://blog.codinghorror.com/code-smells/>

<https://refactoring.com/catalog/>

Caso de Estudo

Sistema de alugueis de filmes do Google Play

Acabaram de chegar

Lançamentos do cinema

See more



Skyscraper
Action & Adventure

★★★★★ R\$25.90



Hotel Transylvania 3
Portuguese audio

★★★★★ R\$16.90



Ant-Man and the Wasp
Action & Adventure

★★★★★ R\$7.90



Jurassic World: Fallen Kingdom
Action & Adventure

★★★★★ R\$6.90

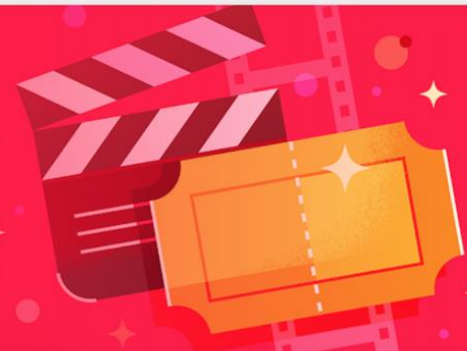


Solo
Action & Adventure

★★★★★ R\$7.90

Alugue qualquer filme por R\$3,90

Clique para resgatar

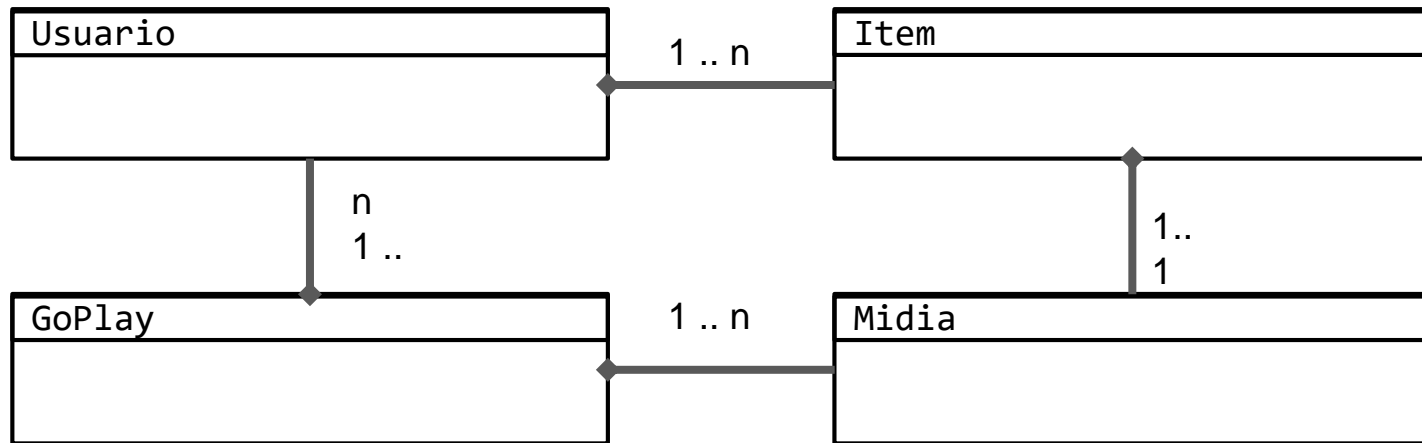


Modelando o Problema

Hands On:

<https://github.com/flaviovd/programacao-2/tree/master/exemplos/aula15-refatoramento/01-codigo-ruim>

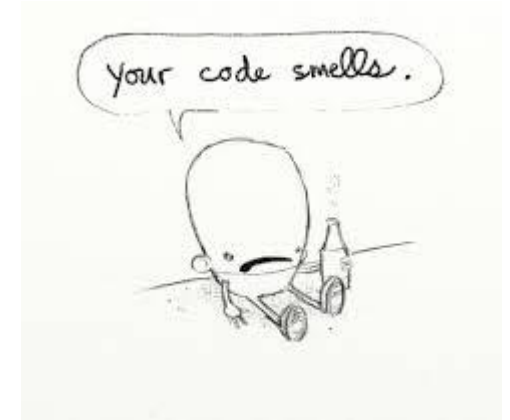
- Acompanhar código do GitHub
- Sistema um pouco complexo para slides



Foco do nosso problema

Um método que computa o total que um usuário gastou na sessão

```
double Usuario::total_gastos() {  
    std::vector<Item>::iterator it = this->_midias.begin();  
    std::vector<Item>::iterator ed = this->_midias.end();  
    double total_gastos = 0.0;  
    for (; it != ed; it++) {  
        Item item = *it;  
        Midia midia = item.get_midia();  
        switch(midia.get_tipo()) {  
            case Tipo::EPISODIO_SERIE:  
                if (item.foi_compra()) {  
                    total_gastos += 12.0;  
                } else {  
                    total_gastos += 3.50;  
                }  
                break;  
            case Tipo::LANCAMENTO:  
                if (item.foi_compra()) {  
                    total_gastos = 32.0;  
                } else {  
                    total_gastos += 16.00;  
                }  
                break;  
            case Tipo::NORMAL:  
                if (item.foi_compra()) {  
                    total_gastos = 15.0;  
                } else {  
                    total_gastos += 7;  
                }  
                break;  
        }  
    }  
    return total_gastos;  
}
```



Bad Smell #1

Método muito longo. Tá na cara. Como Resolver?

```
double Usuario::total_gastos() {  
    std::vector<Item>::iterator it = this->_midias.begin();  
    std::vector<Item>::iterator ed = this->_midias.end();  
    double total_gastos = 0.0;  
    for (; it != ed; it++) {  
        Item item = *it;  
        Midia midia = item.get_midia();  
        switch(midia.get_tipo()) {  
            case Tipo::EPISODIO_SERIE:  
                if (item.foi_compra()) {  
                    total_gastos += 12.0;  
                } else {  
                    total_gastos += 3.50;  
                }  
                break;  
            case Tipo::LANCAMENTO:  
                if (item.foi_compra()) {  
                    total_gastos = 32.0;  
                } else {  
                    total_gastos += 16.00;  
                }  
                break;  
            case Tipo::NORMAL:  
                if (item.foi_compra()) {  
                    total_gastos = 15.0;  
                } else {  
                    total_gastos += 7;  
                }  
                break;  
        }  
    }  
    return total_gastos;  
}
```


Solução: Bad Smell #1

Extração de métodos comuns. Parece que vai ajudar...

```
double Usuario::_preco_serie(Item &item) {  
    if (item.foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}  
  
double Usuario::_preco_lancamento(Item &item) {  
    if (item.foi_compra()) {  
        return 32.0;  
    } else {  
        return 16.00;  
    }  
}
```

Solução: Bad Smell #1

Método menor. Porém ainda temos problemas

```
double Usuario::total_gastos() {
    std::vector<Item>::iterator it = this->_midias.begin();
    std::vector<Item>::iterator ed = this->_midias.end();
    double total_gastos = 0.0;
    for (; it != ed; it++) {
        Item item = *it;
        Midia midia = item.get_midia();
        switch(midia.get_tipo()) {
            case Tipo::EPISODIO_SERIE:
                total_gastos += this->_preco_serie(item);
                break;
            case Tipo::LANCAMENTO:
                total_gastos += this->_preco_lancamento(item);
                break;
            case Tipo::NORMAL:
                total_gastos += this->_preco_normal(item);
                break;
        }
    }
    return total_gastos;
}
```

Solução: Bad Smell #1

Qual o problema aqui?

```
double Usuario::_preco_serie(Item &item) {  
    if (item.foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}  
  
double Usuario::_preco_lancamento(Item &item) {  
    if (item.foi_compra()) {  
        return 32.0;  
    } else {  
        return 16.00;  
    }  
}
```

Bad Smells #2, 3

Parece que Usuário sabe muito sobre os preços

- Note que a classe usuário sabe muito sobre como os preços são computados
- #2: Usuário é uma classe invejosa
- #3: Muita intimidade com o preço

Solução: Bad Smells #2, 3

Vamos mover o conhecimento para o local correto. Item.

```
double Item::_preco_serie() {  
    if (this->foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}
```

```
double Item::get_valor() {  
    switch(this->_midia.get_tipo()) {  
        case Tipo::EPISODIO_SERIE:  
            return this->_preco_serie();  
        case Tipo::LANCAMENTO:  
            return this->_preco_lancamento();  
        case Tipo::NORMAL:  
            return this->_preco_normal();  
    }  
}
```

Solução: Bad Smells #2, 3

Vamos mover o conhecimento para o local correto. Item.

- ❑ Estamos no caminho certo
- ❑ Olhe como o método do usuário fica

```
double Usuario::total_gastos() {  
    std::vector<Item>::iterator it = this->_midias.begin();  
    std::vector<Item>::iterator ed = this->_midias.end();  
    double total_gastos = 0.0;  
    for (; it != ed; it++) {  
        Item item = *it;  
        total_gastos += item.get_valor();  
    }  
    return total_gastos;  
}
```

Solução: Bad Smells #2, 3

Vamos mover o conhecimento para o local correto. Item.

- ❑ A classe nem sequer usa Midia
- ❑ Método curto e direto
- ❑ Vamos limpar a "sujeira" C++

```
double Usuario::total_gastos() {  
    std::vector<Item>::iterator it = this->_midias.begin();  
    std::vector<Item>::iterator ed = this->_midias.end();  
    double total_gastos = 0.0;  
    for (; it != ed; it++) {  
        Item item = *it;  
        total_gastos += item.get_valor();  
    }  
    return total_gastos;  
}
```

Solução: Bad Smells #2, 3

Vamos mover o conhecimento para o local correto. Item.

- ❑ A classe nem sequer usa Midia
- ❑ Método curto e direto
- ❑ Good, but not yet the best.

```
double Usuario::total_gastos() {  
    auto it = this->_midias.begin();  
    auto ed = this->_midias.end();  
    double total_gastos = 0.0;  
    for (; it != ed; it++) {  
        Item item = *it;  
        total_gastos += item.get_valor();  
    }  
    return total_gastos;  
}
```


Solução: Bad Smells #2, 3

Vamos mover o conhecimento para o local correto. Item.

- ❑ A classe nem sequer usa Midia
- ❑ Método curto e direto
- ❑ Melhor

```
double Usuario::total_gastos() {  
    double total_gastos = 0.0;  
    for (auto item : this->_midias) {  
        total_gastos += item.get_valor();  
    }  
    return total_gastos;  
}
```

Bad Smells #4, 5, 6

Os itens agora viraram o problema

- Note que o item agora é um problema
- Bad Smell #4: Código Repetido

```
double Item::_preco_serie() {  
    if (this->foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}
```

Bad Smells #4, 5, 6

Os itens agora viraram o problema

- Note que o item agora é um problema
- Bad Smell #4: Código Repetido
 - Note que o if/else abaixo se repete em três métodos diferentes

```
double Item::_preco_serie() {  
    if (this->foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}
```

Bad Smells #4, 5, 6

Os itens agora viraram o problema

- Note que o item agora é um problema
- Bad Smell #4: Código Repetido
- Bad Smell #5: Classe Longa
 - Os três métodos no estilo abaixo

```
double Item::_preco_serie() {  
    if (this->foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}
```

Bad Smells #4, 5, 6

Os itens agora viraram o problema

- ❑ Note que o item agora é um problema
- ❑ Bad Smell #4: Código Repetido
- ❑ Bad Smell #5: Classe Longa
- ❑ Bad Smell #6: Uso de Magic Numbers

```
double Item::_preco_serie() {  
    if (this->foi_compra()) {  
        return 12.0;  
    } else {  
        return 3.50;  
    }  
}
```

Bad Smell #6

Números mágicos são uma boa dica para refatorar

- Números mágicos são constantes "soltas"
- Veja o exemplo abaixo
 - Qual o significado do número 7?
 - Se tivermos que mudar por outro número?

```
for (int i = 0; i < 7; i++) {  
    cartas.push_back(deck.random());  
}
```

Solução: Bad Smell #6

Uso de constantes, re-organizar dados, move method

- Podemos definir constantes usando mais de uma forma em C++
 - ❏ `#define NUM_CARTAS 7;`
 - ❏ `static int const NUM_CARTAS;`
- ❏ Podemos também re-organizar os dados
 - No caso de estudo, os preços devem ficar junto com os itens
- Criar classes que manipulam os números

Pensando em um sistema real

Itens tem valores fixos, mídias não

- Note que no momento os valores estão atrelados aos itens. Porém note que:
 - Compramos uma única vez
- Parece que não faz sentido a classe item saber computar valores

Item ideal

Mais uma classe pequena =)

- ❑ Precisamos da lógica em algum local
- ❑ A classe Midia é um bom ponto de partida

```
Item::Item(Midia &midia, double valor, bool compra):  
    _midia(midia), _valor(valor), _compra(compra) {}  
  
Midia Item::get_midia() {  
    return this->_midia;  
}  
  
double Item::get_valor() {  
    return this->_valor;  
}
```

Nova Midia

```
Midia::Midia(std::string nome, double preco_aluguel, double preco_compra,
             Tipo tipo):
    _nome(nome), _preco_aluguel(preco_aluguel),
    _preco_compra(preco_compra), _tipo(tipo) {}

Tipo Midia::get_tipo() {
    return this->_tipo;
}

std::string Midia::get_nome() {
    return this->_nome;
}

double Midia::get_preco_aluguel() {
    return this->_preco_aluguel;
}

double Midia::get_preco_compra() {
    return this->_preco_compra;
}
```

Classe GoPlay

Uso de constantes para problemas de magic numbers

```
double const GoPlay::ALUGUEL_NORMAL = 7.0;
double const GoPlay::COMPRA_NORMAL = 15.0;

double const GoPlay::ALUGUEL_LANCAMENTO = 16.0;
double const GoPlay::COMPRA_LANCAMENTO = 32.0;

double const GoPlay::ALUGUEL_SERIE = 3.50;
double const GoPlay::COMPRA_SERIE = 12.0;

GoPlay::GoPlay() {
    this->_codigo_midia = 0;
    this->_codigo_usuario = 0;
}
```

Mais problemas

Parece que a classe GoPlay está com problemas

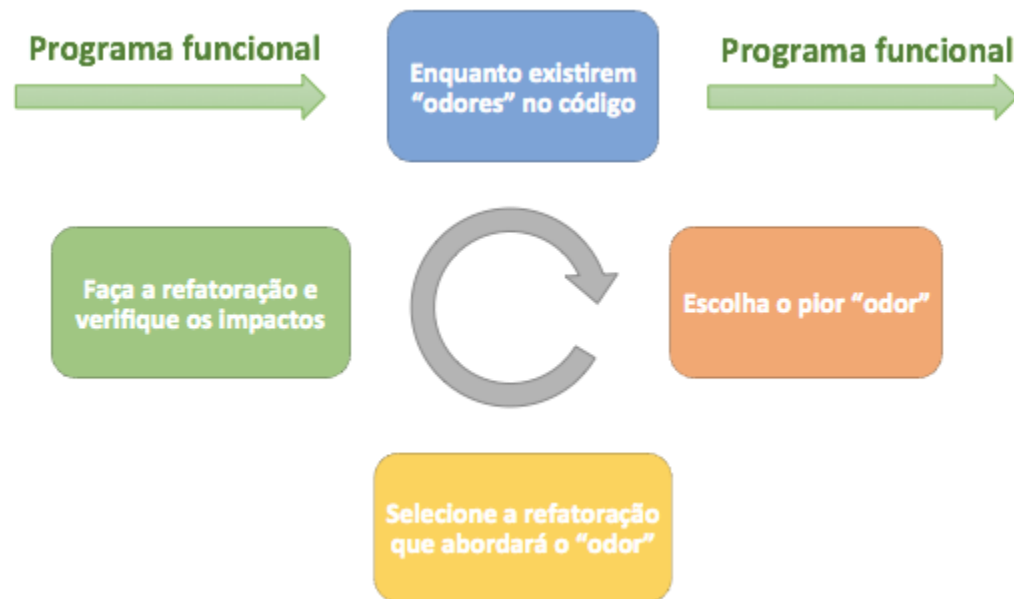
- Temos que limpar o código duplicado
- Porém note que:
 - Para um sistema de Filmes, Séries e Lançamentos
 - Onde o preço não muda ao longo do tempo
 - Estamos ok!
- Keep it simple.

Ainda não estamos prontos

Vamos agora evoluir o programa

□ Novas User Stories

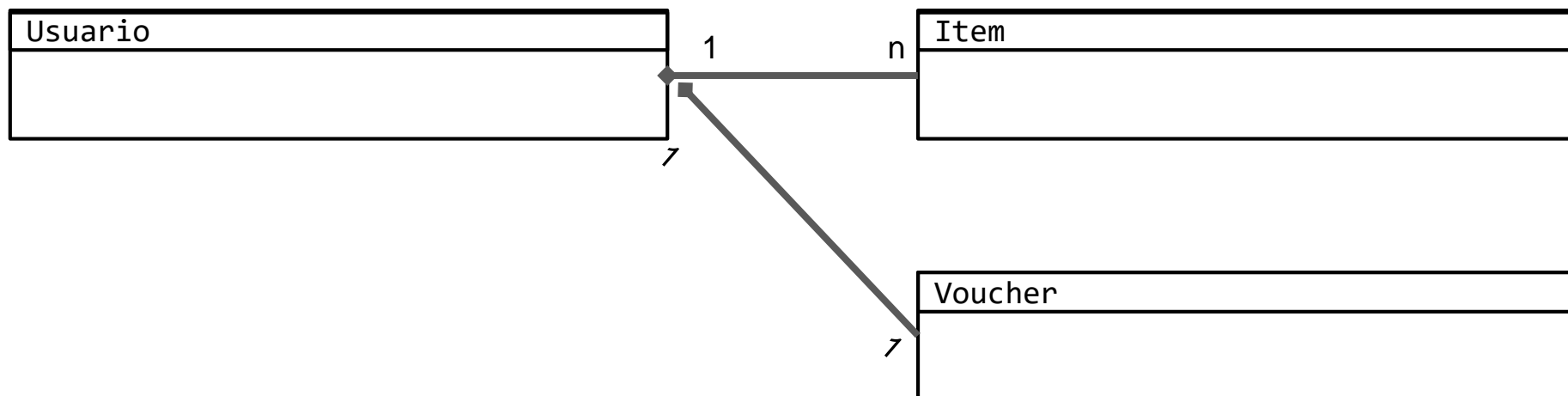
- Preços que mudam com o tempo
- Aplicação de Vouchers de Preços



Sistema de Vouchers

Com o código atual, muda apenas a GoPlay e Usuario. Isto é bom!

- Assumindo que os vouchers colocam um preço único em tudo. Estilo acima.
- Cada usuário pode ter 1 voucher



Sistema de Vouchers

Com o código atual, muda apenas a GoPlay e Usuario. Isto é bom!

- ☐ Mudança ao setar o preço
- ☐ Verificamos se usuário tem voucher
- ☐ Se sim, preço novo
- ☐ Se não, preço antigo

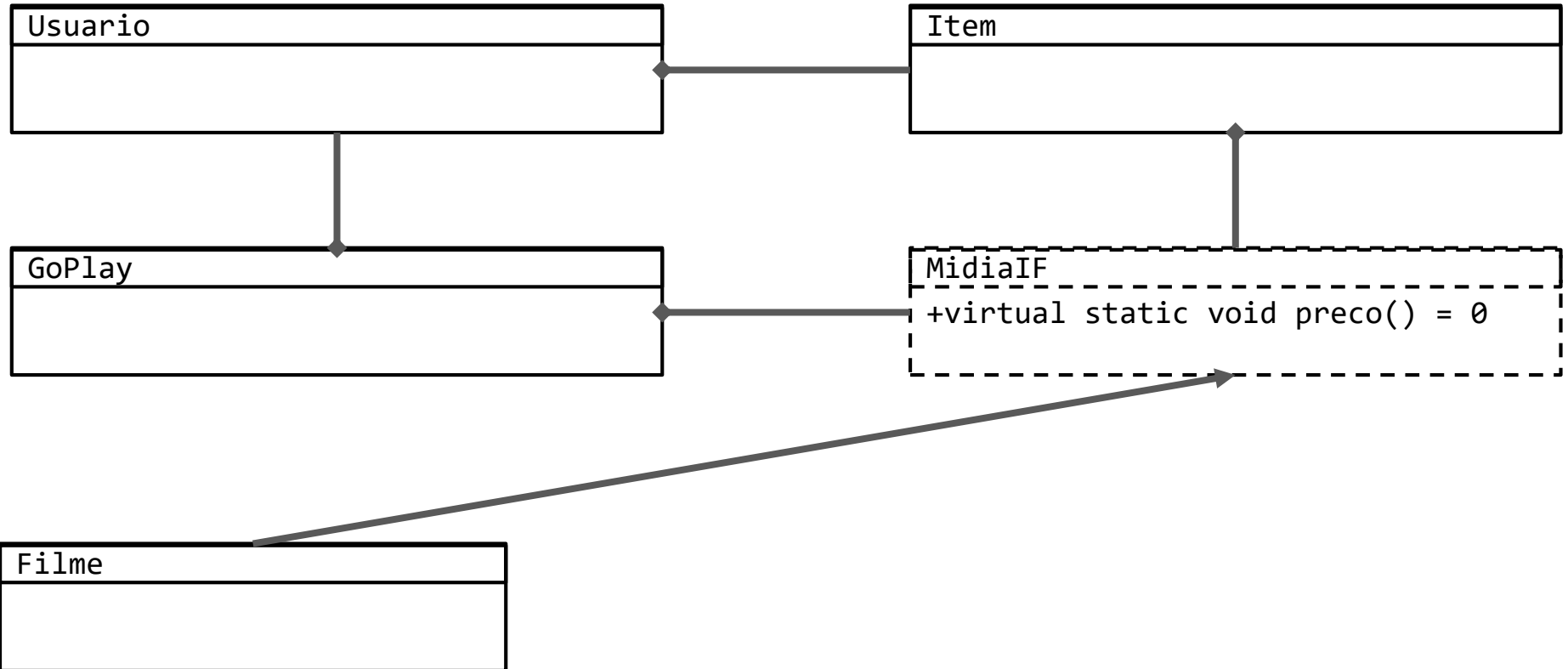
Nova Funcionalidade

Compra e Aluguel de Livros

- Ao invés de Bad Smells vamos adicionar novas Mídias no nosso Google Play
- Como resolver tal caso?
 - Note que não existem séries de livros
 - Pelo menos não no estilo seriados de TV
 - O Enum é um impecilho
 - Além de tal, temos um comportamento comum (os preços)

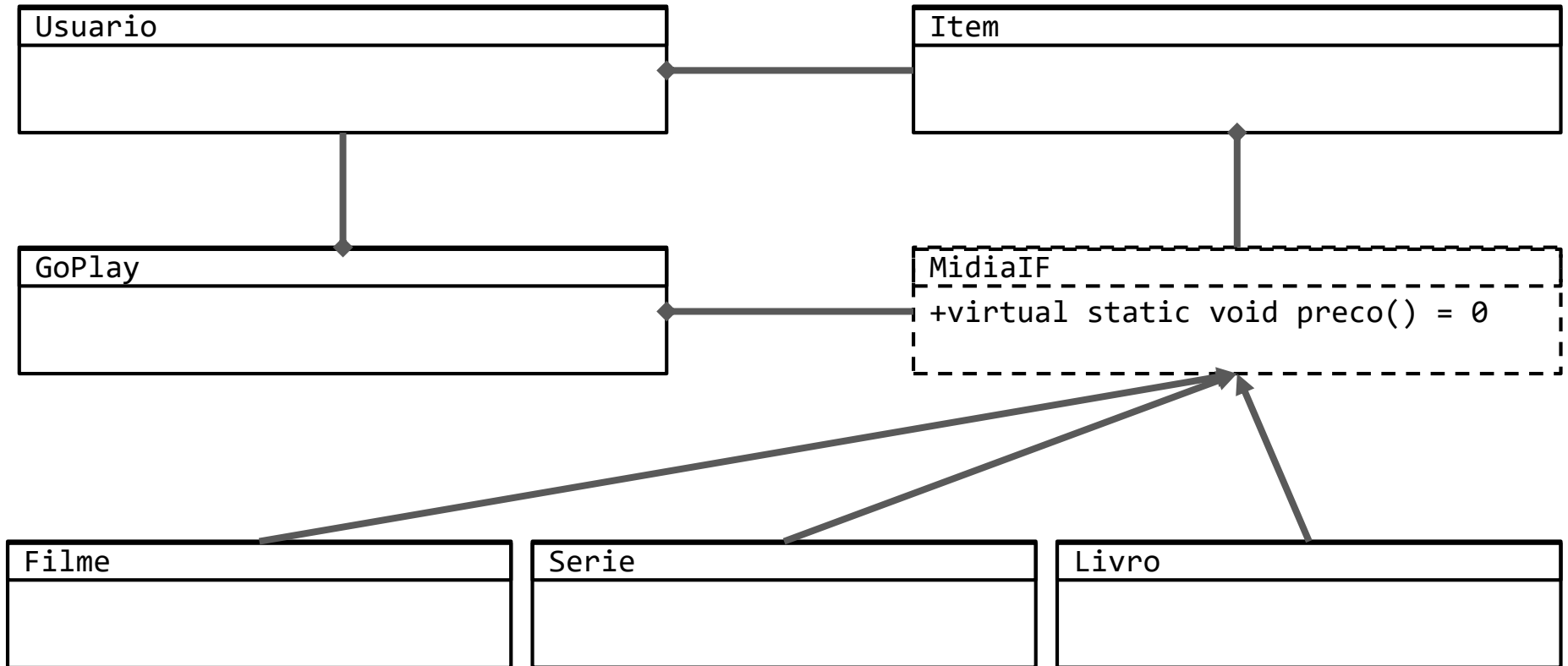
Refatorando: Diagrama Novo

Extraindo uma interface comum. Testar se deu certo



Evoluindo: Diagrama Novo

Depois de refatorar, implementar tipos novos



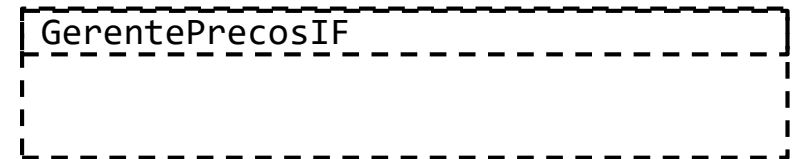
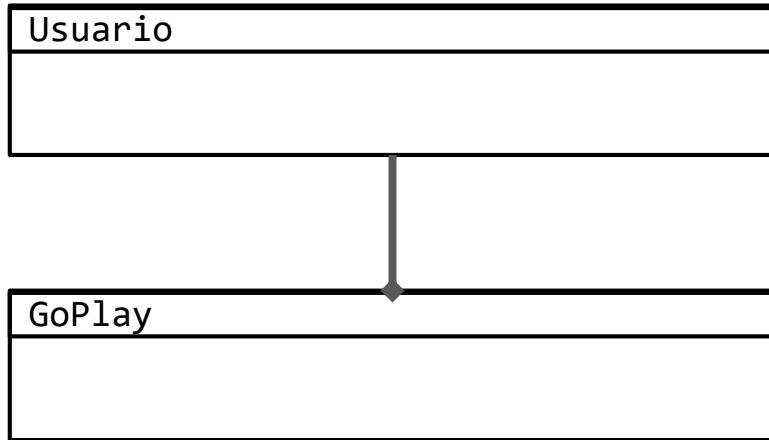
Nova Funcionalidade

Mercado de Preços

- Cada MidiaF vai ter regras diferentes sobre como o preço muda
- Como resolver?

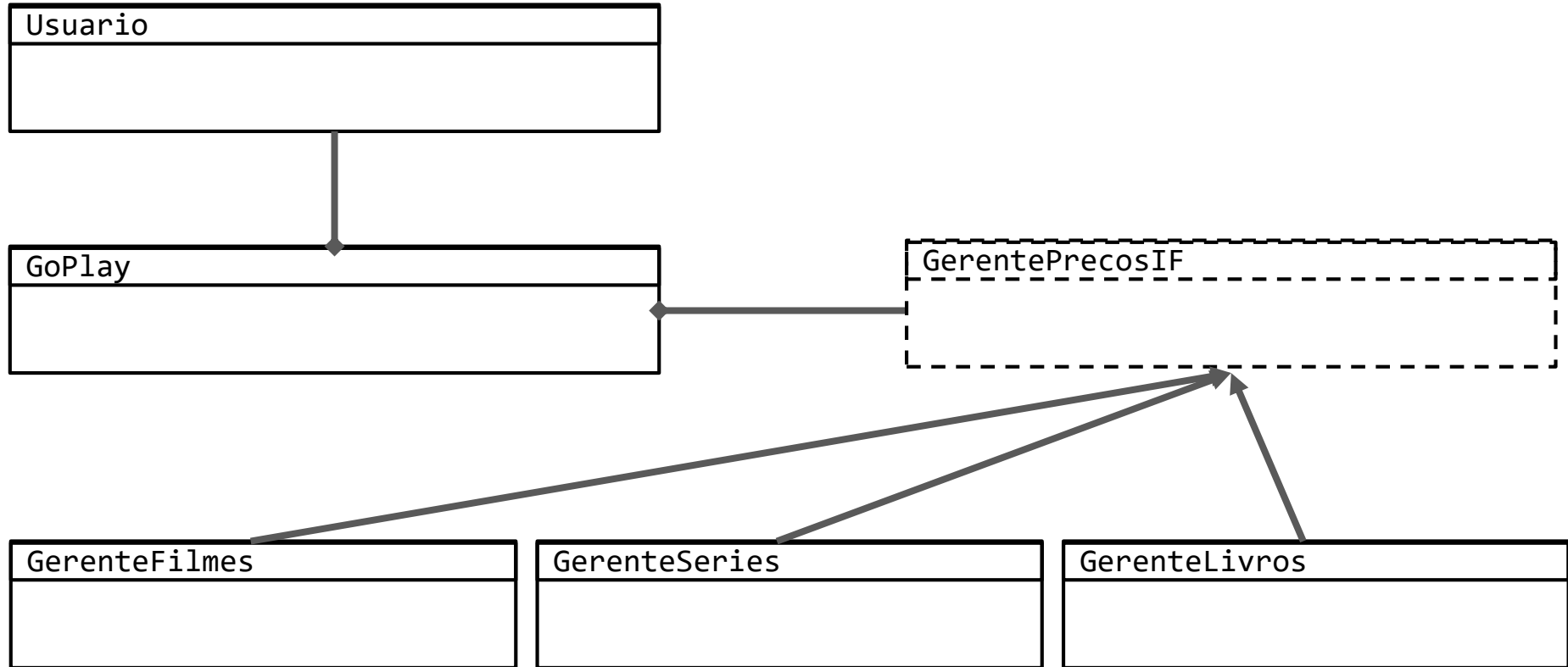
Solução

Refatorando



Ainda temos problemas

Evoluindo



Quando parar?

Keep it simple Stupid!

- Podemos continuar eternamente
- É bom para quando:
 - Nossas user stories são cumpridas
 - Iniciar novamente a partir de cada funcionalidade nova
 - Não faça over-designs desde o início
- Bons programadores escrevem código para outros programadores (humanos).

Considerações finais

- Benefícios desenvolvedores
 - Melhora manutenção
 - Aumenta reusabilidade
 - Facilita testes automatizados
 - Alta coesão, baixo acoplamento
 - Maior robustez do código
 - Facilita o trabalho em equipe

Considerações finais

- Benefícios negócios
 - Baixo índice de erros
 - Código de maior valor
 - Facilidade de adaptação de requisitos
 - Fácil adicionar novas features
 - Produto liberado mais rapidamente
 - Atacar desempenho/segurança

Programação e Desenvolvimento de Software 2

TDD

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Metodologias de Desenvolvimento

- ❑ Quando devemos iniciar a fase de testes?
 - ❑ Depois de terminada a codificação?
 - ❑ Não é uma boa ideia! Por que?
- ❑ Desenvolvimento Orientado por Testes (TDD)
 - ❑ Foco deve ser no requisito, não no código!
 - ❑ Interface → Comportamento

Test Driven Development (TDD)

- Escrevemos os testes antes do código
 - Antes?!
- Prática extremamente recomendada
 - Código se adapta ao teste, não o contrário!
- Abordagem incremental
 - Pequenos passos (testes) ajudam a alcançar um resultado final de qualidade (projeto)
- Maneira de se desenvolver, não de testar!

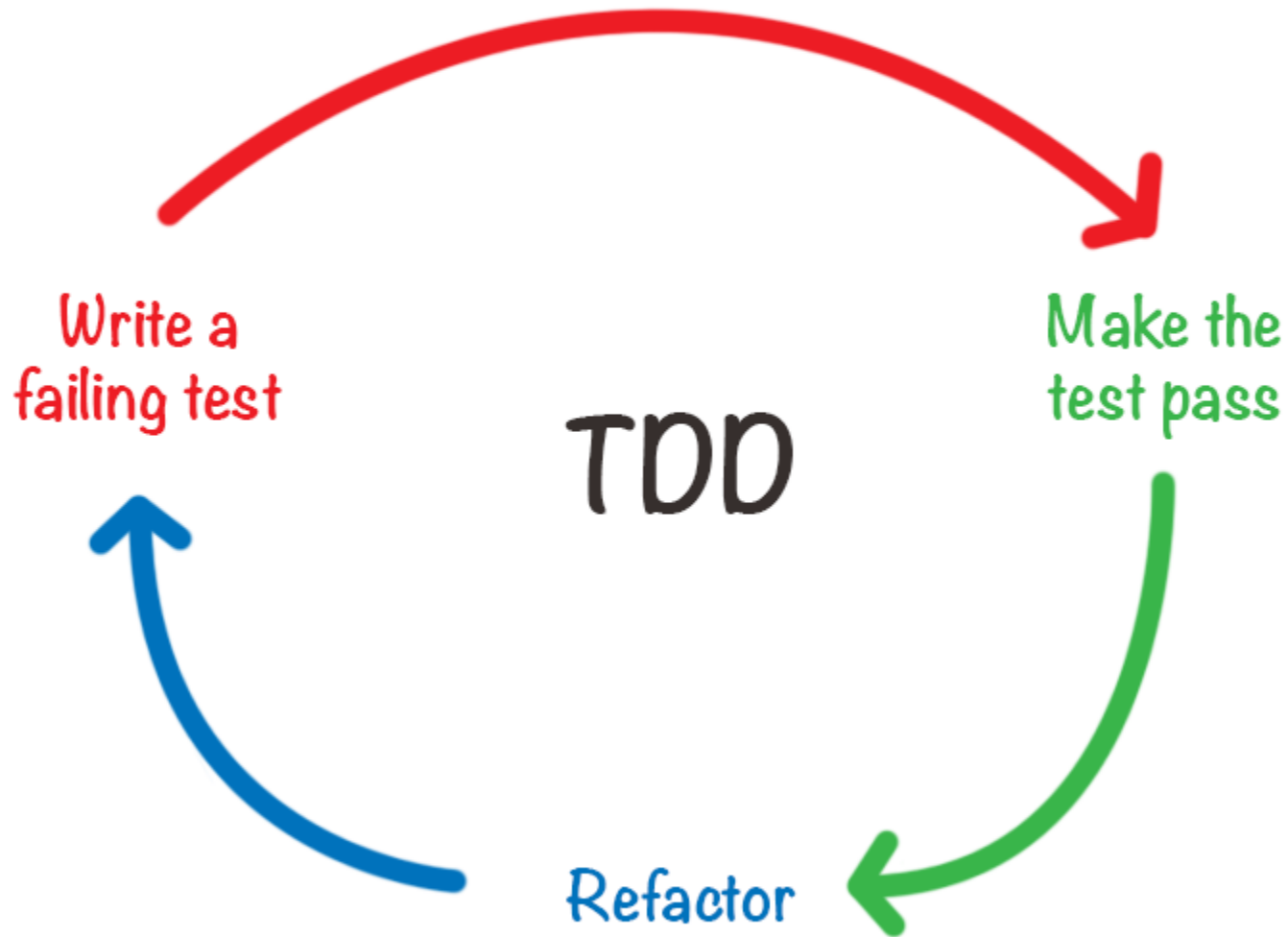
TDD: Implicações

- Desenvolvimento orgânico, onde o código em execução gera o retorno necessário para tomar as decisões que orientam o próprio desenvolvimento
- Testes implementados pelos próprios desenvolvedores

TDD: Implicações

- ❑ O ambiente de desenvolvimento deve fornecer respostas rápidas para pequenas mudanças
- ❑ O projeto deve ter alta coesão, componentes fracamente acoplados, exatamente para permitir testar facilmente

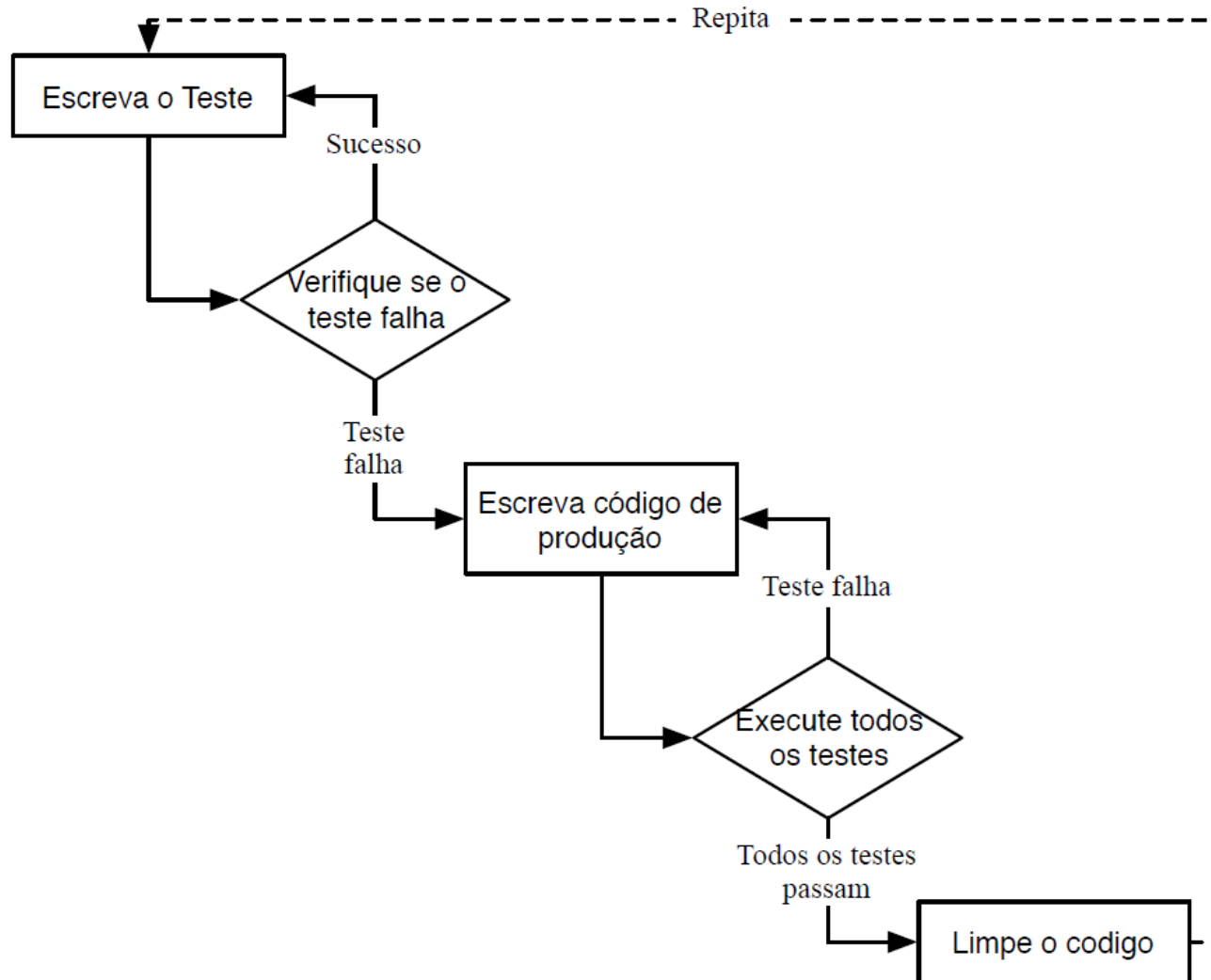
Ciclo de Desenvolvimento



Test Driven Development (TDD)

- Escreva um teste que falha
- Faça o teste passar rapidamente
- Refatore

Estágios



Estágios

- ❑ Escreva um teste simples
- ❑ Compile. Ele não deve compilar uma vez que o código ainda não foi escrito
- ❑ Implemente apenas o código necessário para fazer o teste compilar
- ❑ Execute o teste e veja que ele vai **falhar**

Estágios

- ❑ Implemente o código necessário para fazer o teste passar
- ❑ Execute e veja o teste **passar**
- ❑ **Refatore** o código para torná-lo mais claro
- ❑ Repita

Testes devem seguir modelo FIRST

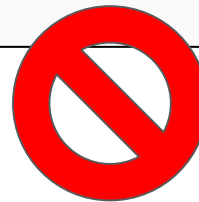
- ❑ F (Fast) - Rápidos: devem ser rápidos, pois testam apenas uma unidade;
- ❑ I (Isolated) - Testes unitários são isolados, testando individualmente as unidades e não sua integração;
- ❑ R (Repeateble) - Repetição nos testes, com resultados de comportamento constante;
- ❑ S (Self-verifying) - A auto verificação deve verificar se passou ou se deu como falha o teste;
- ❑ T (Timely) - O teste deve ser oportuno, sendo um teste por unidade.

TDD: Exemplo Fatorial

□ Escrevendo um teste simples

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("Testando o fatorial") {
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(4) == 24);
    CHECK(fatorial(10) == 3628800);
}
```



Não compila pois não existe o método fatorial!

TDD: Exemplo Fatorial

- Implemente apenas o código necessário para compilar

```
#include "fatorial.h"

int fatorial(int n) {
    return n;
}
```

TDD: Exemplo Fatorial



❑ ○ teste vai **falhar**

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options
=====
testes.cpp:4:
TEST CASE: Testando o fatorial

testes.cpp:6: ERROR: CHECK( fatorial(3) == 6 ) is NOT correct!
  values: CHECK( 3 == 6 )

testes.cpp:7: ERROR: CHECK( fatorial(4) == 24 ) is NOT correct!
  values: CHECK( 4 == 24 )

testes.cpp:8: ERROR: CHECK( fatorial(10) == 3628800 ) is NOT correct!
  values: CHECK( 10 == 3628800 )

=====
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:     4 |      1 passed |      3 failed |
[doctest] Status: FAILURE!
```

TDD: Exemplo Fatorial

- Implemente o código necessário para fazer o teste passar

```
#include "fatorial.h"

int fatorial(int n) {
    if (n <= 1) return n;
    return n * fatorial(n-1);
}
```

TDD: Exemplo Fatorial

☐ O teste passou

```
[doctest] doctest version is "2.0.1"
[doctest] run with "--help" for options

=====
[doctest] test cases:      1 |      1 passed |      0 failed |      0 skipped
[doctest] assertions:     4 |      4 passed |      0 failed |
[doctest] Status: SUCCESS!
```

☐ Refatore

☐ Repita

TDD: Exemplo Fatorial

☐ Escrevendo outro teste simples

☐ Caso especial: zero!

```
#include "doctest.h"
#include "fatorial.h"

TEST_CASE("1: fatorial of 0 is 1 (corner case)") {
    CHECK(fatorial(0) == 1);
}

TEST_CASE("2: fatorials of 1 and higher are computed (caso geral)") {
    CHECK(fatorial(1) == 1);
    CHECK(fatorial(2) == 2);
    CHECK(fatorial(3) == 6);
    CHECK(fatorial(10) == 3628800);
}
```

☐ E assim por diante...

Leis do TDD

- Primeira lei: você não deve escrever qualquer implementação antes que você tenha escrito um teste que falhe;
- Segunda lei: você não deve escrever mais que um teste unitário para demonstrar uma falha;
- Terceira lei: você não deve escrever mais do que o necessário para passar por um teste que está falhando.

Vantagens

□ Aprendizado

- Quanto maior o tempo entre a inserção do erro no código e a descoberta do mesmo pelo desenvolvedor maior a probabilidade dele repetir-se;
- Se o erro é “descoberto” alguns segundos após ser introduzido, ele é corrigido rapidamente, e o desenvolvedor aprende com isso, passando a codificar melhor.

Vantagens

- Redução de custos e vulnerabilidades do sistema
 - tempo entre a inserção do bug e a detecção é o que torna a depuração custosa
 - o teste expõe o erro assim que ele entra no sistema, o que evita muita perda de tempo com depurações demoradas
 - programação defensiva (veremos nas próximas aulas)

Vantagens

- Aumento da qualidade e da produtividade
 - detectando uma falha rapidamente...
 - evitamos longas sessões de depuração que costumam tomar boa parte do tempo dos projetos
 - com mais tempo disponível, podemos aprender mais, e desenvolvedores podem codificar mais rapidamente e com mais qualidade
 - ou seja, aumenta-se a produtividade e reduz-se a incidência de defeitos (i.e. qualidade)

Vantagens

- Aprimoramento do projeto (arquitetura do código melhora)
 - Escrever um caso de teste completo força a criação de um código desacoplado (não vinculado estritamente a outro código), aumentando assim a sua coesão e diminuindo seu acoplamento
 - Mais modular, flexível e extensível

Vantagens

☐ Documentação

- ☐ Um caso de teste de unidade bem escrito proporciona uma especificação de implementação e comunica a finalidade do código de forma clara.
- ☐ Sempre que o código for alterado, o caso de teste de unidade deve ser atualizado para passar pelo conjunto de testes
 - ☐ Sincronismo entre teste e código

Vantagens

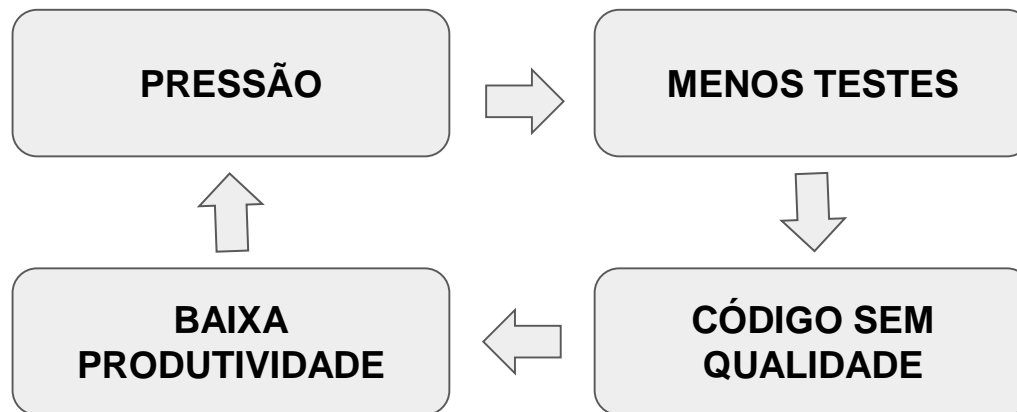
- Mudanças seguras no código
 - O TDD fornece feedback contínuo quanto ao resultado das mudanças do código em outras partes do sistema
- Código nítido
 - significa que sua finalidade é expressa com clareza, pode ser alterado para receber novos recursos e não tem duplicação

Test Driven Development (TDD)

- Até o momento a maioria dos VPLs seguiram uma abordagem TDD
 - Sem o refactor
- Temos os testes antes
- Depois nos preocupamos com a classe em si. O teste guia o código da classe

Test Driven Development (TDD)

- Programadores sabem que devem escrever o teste antes, mas são poucos o que fazem
- Gera um ciclo vicioso!



Test Driven Development (TDD)

- ❑ Exige comprometimento da equipe!
- ❑ Não garante o sucesso do projeto
 - ❑ Os testes foram bem feitos?
 - ❑ Robustos, fácil manutenção, realmente testado,
- ❑ Refatoração é muito importante para melhoria da qualidade

Exercício

- Como testar peças de xadrez? Isto é, temos uma interface `Peca` com o método `boolean podeMover(int x, int y);`
- Diversas peças implementa a mesma.
- Uma classe `Tabuleiro` com um método `void move(Peca &peca, int x, int y);`

Exercício

- ❑ Precisamos de testes para cada classe que implementa a interface Peca
- ❑ Além de um teste para Tabuleiro
 - ❑ O mesmo que conhece o tamanho

Programação e Desenvolvimento de Software 2

Programação Defensiva

Prof. Julio Cesar S. Reis
julio.reis@dcc.ufmg.br

Introdução

☐ Direção defensiva

Direção Defensiva é o ato de conduzir de modo a evitar acidentes, apesar das ações incorretas (erradas) dos outros e das condições adversas (contrárias), que encontramos nas vias de trânsito.

– DETRAN

☐ Desenvolvimento de software

- ☐ Queremos evitar acidentes (erros)
- ☐ Apesar de ações incorretas (usuários)
- ☐ Em condições adversas (resto do programa)

Introdução

- ❑ Programação defensiva
 - ❑ Não é ser defensivo sobre sua programação
 - ❑ "Eu garanto que funciona!"
- ❑ "Garbage in, garbage out"
 - ❑ Entradas ruins produzem saídas ruins
 - ❑ Colocar a culpa (obrigação) no usuário?
- ❑ Bons programadores
 - ❑ Se defendem, estilo motoristas

Programação Defensiva

- Forma de design protetivo destinado a garantir o funcionamento contínuo de um software sob circunstâncias não previstas
 - “Garbage in, nothing out”
 - “Garbage in, error message out”
 - “No garbage allowed in”
- Erros mais fáceis de encontrar e corrigir e menos prejudiciais ao código de produção

Programação Defensiva

Robustez vs Corretude

☐ Robustez

- ☐ Sempre tentar fazer algo que permita que o software continue operando, mesmo que isso às vezes leve a resultados imprecisos

☐ Corretude (exatidão)

- ☐ Nunca retornar um resultado impreciso
- ☐ Não retornar nenhum resultado será melhor do que retornar um resultado incorreto
- ☐ Qual característica deve ser priorizada?

Programação Defensiva

Estratégias

- ☐ Validação das entradas
- ☐ Asserções
- ☐ Programação por contrato
- ☐ Barricadas
- ☐ Tratamento de exceções (vimos nas últimas aulas)

Validação das Entradas

☐ Entradas

- ☐ Sem controle, inesperadas, imprevisíveis
- ☐ Podem ser inclusive mal-intencionadas
- ☐ Assuma o pior de todas as entradas!

☐ Tratamento Geral

- ☐ Defina o conjunto de valores de entrada válidas
- ☐ Ao receber uma entrada, valide com esse conjunto
- ☐ Estabeleça um comportamento caso incorreta
 - ☐ Terminar / Repetir / Alertar

Validação das Entradas

Deve ser feita para todo método - Exemplo

- ☐ Quais valores fazem sentido o método receber?

Quais tipos?

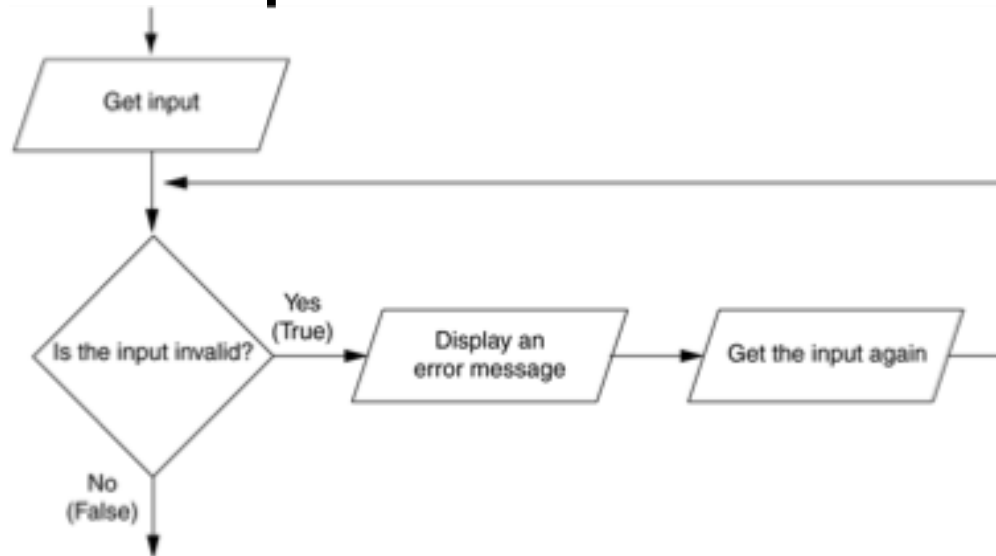
```
Conta c;  
c.depositar('a');  
c.depositar(-10);  
c.depositar(9.99999);  
c.depositar(1.0e-10);
```

- ☐ O valor é numérico?
- ☐ Aceita valores negativos?
- ☐ Número de casas decimais importa?
- ☐ Deve ser composto apenas de números?
- ☐ A quantidade é válida? (muito grande ou pequena)

Validação das Entradas

No caso do usuário

□ Validation loop



□ Cada aplicação define tal comportamento

□ Podemos ficar eternamente no loop

□ Sugerir respostas

Asserções

- ☐ Asserção/Assertiva
 - ☐ Predicado inserido para verificar (certificar) que determinada condição (suposição) é verdadeira
- ☐ Argumentos
 - ☐ Expressão booleana (que deve verdadeira)
 - ☐ Mensagem a ser exibida caso não seja
- ☐ Podem ser utilizadas no teste ou código
- ☐ Códigos altamente robustos
 - ☐ Asserções (desenvolvimento) → Exceções (execução)

Asserções vs. Exceções

☐ Asserções

- ☐ Erros fatais (não podem ser tratados)
- ☐ Sempre indicam algum erro no código
- ☐ Situações que nunca deveriam ocorrer

☐ Exceções

- ☐ Situações excepcionais (podem ser tratadas)
- ☐ Podem ocorrer mesmo em códigos corretos
 - ☐ Falta de memória, erro de comunicação, ...
- ☐ Na dúvida, faça uso de exceções

Asserções

Possíveis verificações

- ☐ Parâmetro de entrada está dentro do intervalo esperado
- ☐ Ponteiro a ser utilizado não é NULL
- ☐ Container está vazio (ou preenchido) quando uma rotina começa a ser executada (ou quando termina)
- ☐ Container usado em uma rotina deve/pode conter pelo menos (no máximo) um número de X de elementos
- ☐ Arquivo ou stream está aberto (fechado) quando uma rotina começa a ser executada (termina a execução)

Asserções

Exemplo em Código

```
//#define NDEBUG
#include <iostream>
#include <cassert>

int main() {
    int vetor[10];
    for (int i = 0; i < 15; i++) {
        assert(0 <= i && i < 10);
        vetor[i] = i;
        std::cout << vetor [i] << std::endl;
    }
}
```

Assertões

Código vs. Testes

- ❑ Nos testes usamos assertões para testar nossas classes como clientes dela
- ❑ Dentro do código usamos assertões para indicar erros em tempo de execução
 - ❑ Não pegamos assertões
 - ❑ Não são tipos Exception
 - ❑ Geralmente desabilitada ao lançar o código
- ❑ Fail-fast programming

Asserções

Fail-fast programming

- ☐ Quando ocorrer um problema, o sistema deve falhar imediatamente e visivelmente
 - ☐ Evitar postergar uma falha para o futuro
- ☐ Ajudam a detectar erros precocemente
 - ☐ Análogas a fusíveis em um circuito
 - ☐ Falhas antes que mais danos sejam causados
 - ☐ Também ajudam a identificar a raiz de uma falha
- ☐ Asserções são desativadas no release (!)

Asserções

Uma boa prática é usar o assert em métodos private

```
#include <cassert>
#include <map>
#include <string>
#include <vector>

class ProcessadorTexto {
private:
    // guarda termo -> número de vezes em um documento
    std::map<std::string, std::vector<int>>> _contagens;

    int _soma_um(std::string str) {
        assert(_contagens.count(str));
        int soma = 0;
        for (int count : _contagens[str]) {
            soma += count;
        }
        return soma;
    }
public:
    int soma_total() {
        int soma = 0;
        for (auto pair : _contagens) {
            soma += _soma_um(pair.first);
        }
        return soma;
    }
};
```

Programação por Contrato

- Acordo entre duas ou mais partes
 - Funções devem ser vistas como um contrato
 - Executam uma tarefa específica
 - Não devem fazer outra coisa além disso
 - Produzem alguma saída como resultado
- Métodos públicos definem como nossas classes serão utilizadas

Programação por Contrato

□ Perguntas

- O que o contrato espera?
- O que o contrato garante?
- O que o contrato mantém?

□ Elementos (formalização lógica)

- {Pré-condições} ação {Pós-condições}
{Invariantes}

Exemplo

□ Quão bom é o código abaixo?

```
class Conta {  
private:  
    int _agencia;  
    int _numero;  
    double _saldo;  
public:  
    void sacar(double valor) {  
        this->_saldo -= valor;  
    }  
};
```


Programação por Contrato

Conta Corrente

☐ Pré-condição

- ☐ O que deve ser verdadeiro para a rotina poder ser chamada (requisitos mínimos)

- ☐ Ex.: Saldo em conta

☐ Pós condição

- ☐ O que deve ser verdadeiro após a rotina executar

- ☐ Ex.: Redução do saldo **apenas** se houver saldo suficiente para realizar o saque ou exceção caso contrário

☐ Invariante

- ☐ Condições que devem sempre ser verdade, antes, durante e após a execução de uma região

- ☐ Ex.: Conta corrente sempre com saldo positivo

Programação por Contrato

- Utilize asserções para documentar e verificar pré-condições, pós-condições e invariantes
- E se não for possível cumprir o contrato?
 - Definir um valor de erro global
 - Retornar um valor indicativo (inválido)
 - NULL? False? Número negativo?
 - Lançar uma exceção

Programação por Contrato

- ☐ Pré-condições/Pós condições vs. Herança?
- ☐ Subcontratação
 - ☐ A definição de uma subclasse significa uma extensão do contrato da superclasse
 - ☐ O contrato herdado pode ser redefinido desde que não viole o contrato da superclasse
- ☐ Pode-se enfraquecer as pré-condições e fortalecer as pós-condições de métodos
 - ☐ Condição mais fraca é menos restrita
 - ☐ Condição mais forte é mais restrita

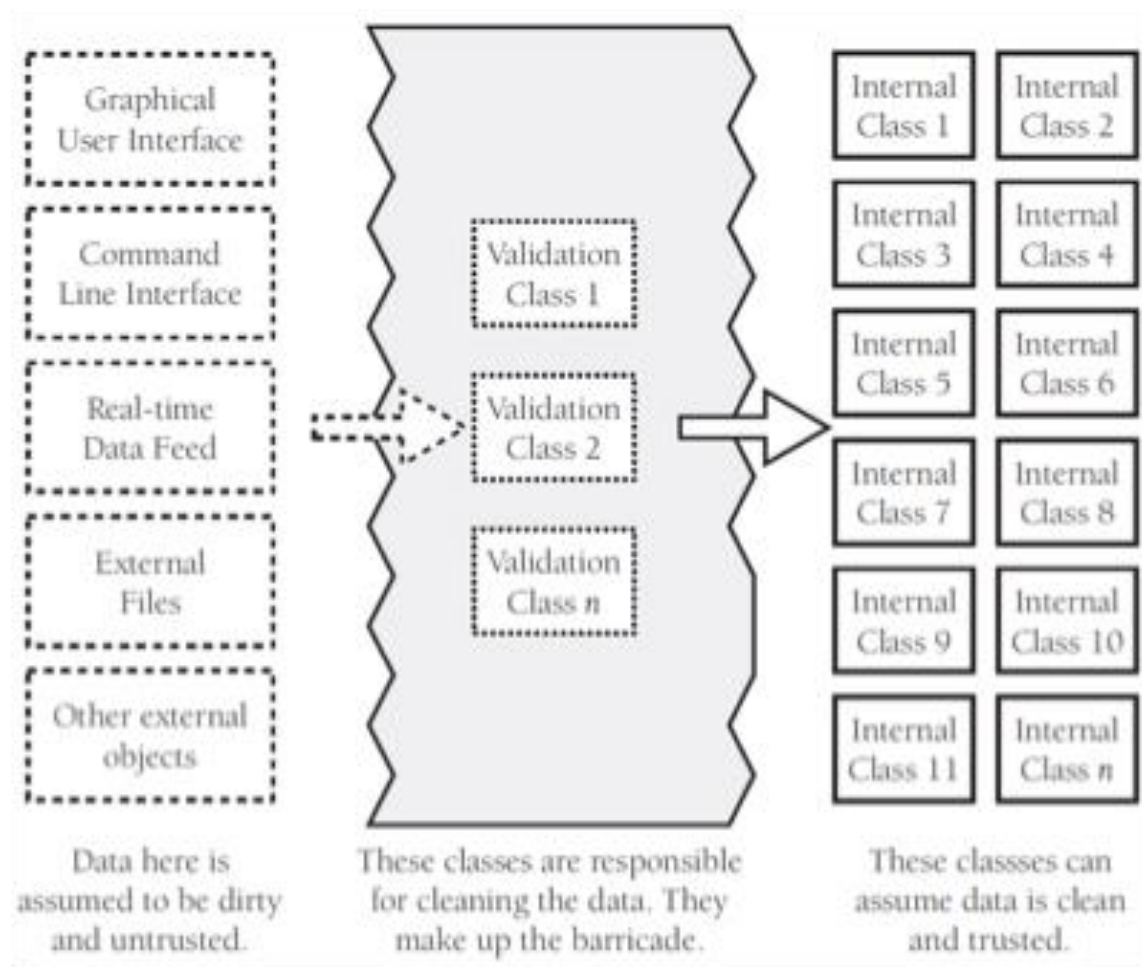
Barricadas

Garantem que o código está correto depois de um certo ponto

- ☐ Crie barricadas no programa para minimizar o dano causado por dados incorretos
- ☐ Interfaces como limites para áreas “seguras”
 - ☐ Verifique todos os dados que cruzam os limites de áreas seguras
- ☐ Partes que funcionam com dados *sujos* e algumas que funcionam com dados *limpos*
 - ☐ Responsabilidade pela verificação centralizada

Barricadas

Caso de Uso: Programa com diversas interfaces



Barricadas

Classes

- ❑ Métodos públicos
 - ❑ Validam os dados externos
- ❑ Métodos privados
 - ❑ Assumem que é seguro usar os dados
- ❑ Asserções vs. Exceções
 - ❑ Considere o uso de exceções para métodos públicos e asserções para métodos privados

TDD e Programação Defensiva

Caminham juntos

- Test driven development
 - Metodologia de desenvolvimento
 - Teste, código, refactor
- Programação defensiva
 - Boas práticas para ter um código seguro
- TDD **deve** verificar as condições da programação defensiva

Considerações Finais

□ Regras gerais

- Nunca assuma nada como verdade absoluta
 - Entradas, comportamento do usuário, recursos, ...
- Utilize padrões pré-estabelecidos
 - Codificação, design, documentação, ...
- Mantenha o código o mais simples possível
 - Deve conter apenas os recursos de que precisa
 - Complexidade é uma ótima fonte de erros
 - Planejamento adequado é essencial!

Considerações Finais

- “Being Defensive About Defensive Programming”
- Críticas
 - Evitar o uso excessivo de programação defensiva
 - Desperdício de tempo e dinheiro
 - Proteção de erros que nunca serão encontrados
 - Tente encontrar o equilíbrio
 - Aplicação → Robustez vs. Corretude

Considerações Finais

- Quanto de programação defensiva deixar no código de produção?
 - Remova o código que resulta em falhas graves
 - Deixar código que verifica erros importantes
 - Mensagens de erro devem ser informativas
 - Registre (log) possíveis falhas (análise posterior)