

Programação e Desenvolvimento de Software 2

Tipos Abstratos de Dados (TADs) e Modularização

Prof. Luiz Chaimowicz
(slides adaptados do Prof. Douglas Macharet)

Introdução

- Algoritmo
 - Sequência de ações executáveis
 - Transformam uma entrada em uma saída
 - Trabalham sobre estruturas de dados
- Estruturas de dados
 - Armazenam as informações
 - Abstração da realidade
 - Suportam as operações dos algoritmos

Introdução

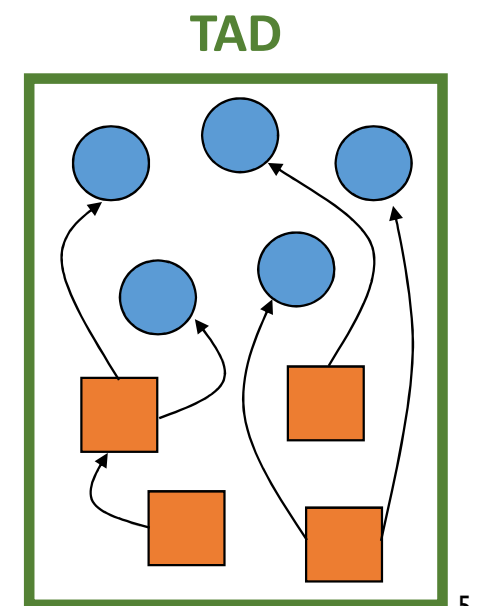
- Diferença entre Programa e Algoritmo?
 - Um programa é uma realização concreta de um algoritmo abstrato, baseado em representações de dados específicas
 - Programas precisam ser implementados numa linguagem que pode ser entendida e seguida pelo computador
 - C, C++, Java, Python, ...

Introdução

- Quando um programa é considerado “bom”?
 1. Funciona (faz o que foi especificado)
 2. Fácil de entender e modificar
 3. Razoavelmente eficiente (recursos)
- Bons programas fazem uso de Abstração
 - Conceito (ideia) → Implementação (concreto)
 - Identificação e definição dos tipos específicos
- Tipos Abstratos de Dados
 - Ajudam fazer (2), que facilita alcançar (1)

Tipos Abstratos de Dados (TADs)

- Agrupa a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados
- **Encapsula** diferentes elementos (membros)
 - Conjunto de variáveis: **Atributos**
 - Conjunto de operações: **Métodos**
- Usuário do TAD x Programador do TAD
 - Usuário só “enxerga” a interface, não a implementação
 - Acesso via as operações disponibilizadas



Tipos Abstratos de Dados (TADs)

- Desvincular a especificação da sua implementação
 - Ocultação de informação (*information hiding*)
 - O usuário pode abstrair da implementação específica
 - Qualquer modificação nessa implementação fica restrita ao TAD
- **Especificação** (interface, contrato)
 - O que esse tipo de “coisa” representa?
 - Quais operações podem ser feitas com ela?
- **Implementação** (código)
 - Como essa “coisa” deve ser implementada?
 - Como as operações funcionam internamente?

Tipos Abstratos de Dados (TADs)

- Lista de números: 4, 1, 5, 6, ...
 - Quais operações podemos ter?
 - Criar uma nova lista, inserir um número, remover, ordenar, ...
 - Como deve ser feita a implementação?

Usuário do TAD

```
int main() {  
    Lista *l;  
    int x = 20;  
  
    l->inserir(x);  
    ...  
}
```

Programador do TAD

- Vetor:

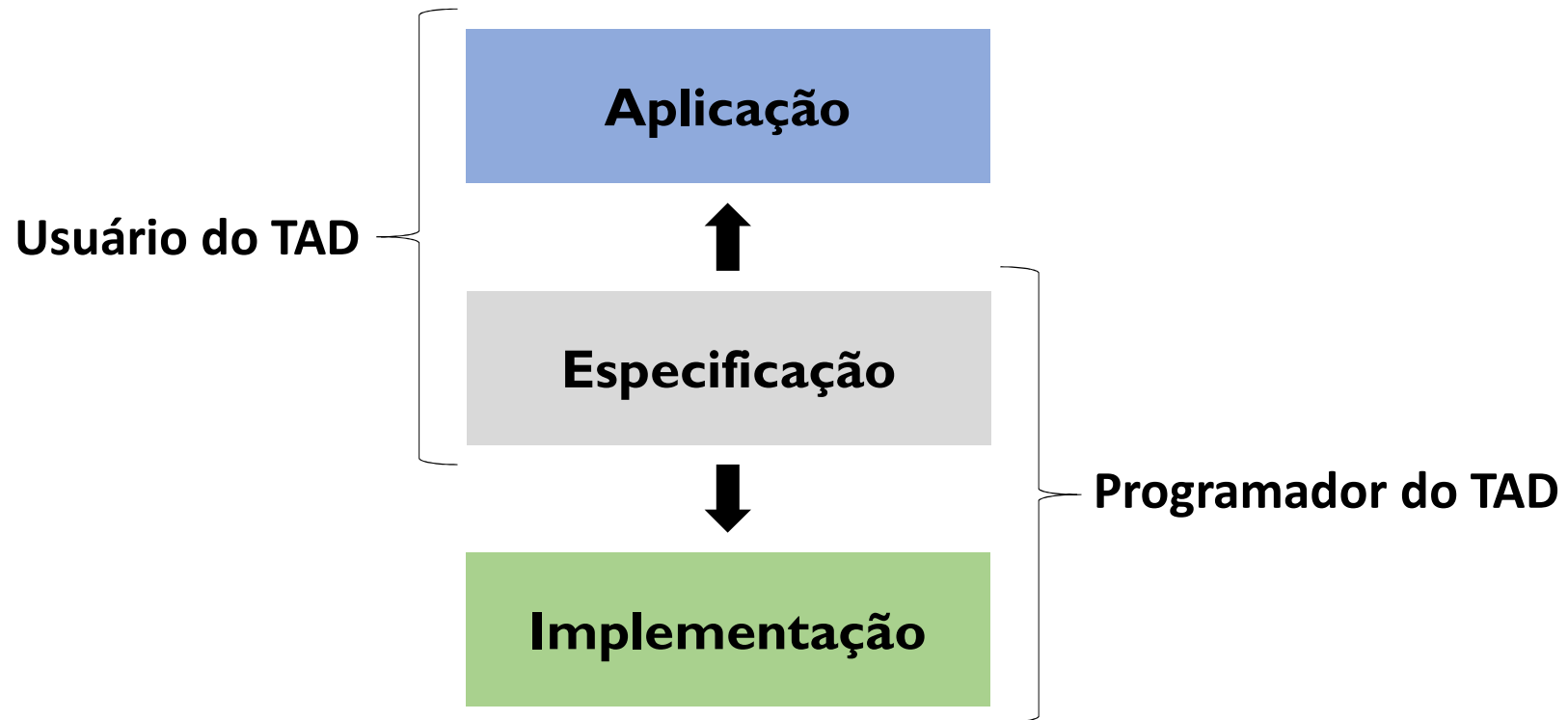
20	13	02	30
----	----	----	----

```
void inserir(int x) {  
    l->arranjo[...] = x;  
    ...  
}
```

- Lista: 

```
void inserir(int x) {  
    Celula *c = criar_celula(x);  
    l->ultimo = c;  
}
```

Tipos Abstratos de Dados (TADs)



Tipos Abstratos de Dados (TADs)

- Um TAD é um tipo caracterizado não apenas pelos valores que o compõe, mas também pelas operações que se aplicam sobre ele.

O que o TAD **representa e faz** é mais importante do que **como** ele faz!

Tipos Abstratos de Dados (TADs)

- Quais as vantagens disso?
 - Integridade, manutenção, reutilização, ...
 - Podemos modificar a implementação interna do TAD sem que o usuário do TAD também precise fazer alguma alteração
- Exemplos
 - String, Ponto3D, Aluno, Carro
 - Lista, Árvore, Pilha, Fila

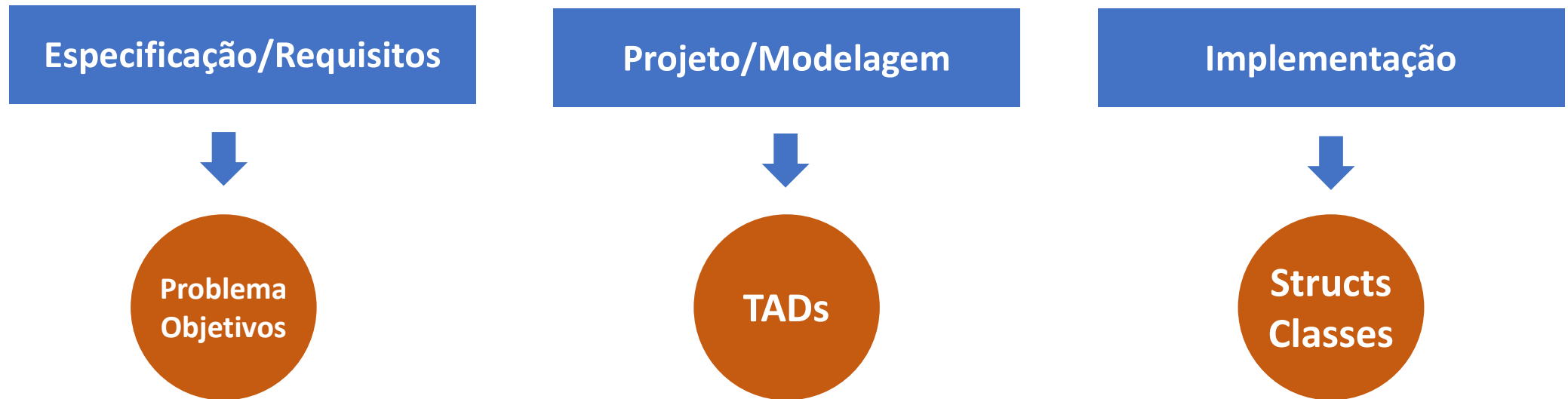
Tipos Abstratos de Dados (TADs)

Operações

- Quais operações devem representar um TAD?
- Filosofia egoísta
 - Se algo não é útil, então não é necessário
 - Informe ao usuário apenas o que ele precisa
 - Lembre-se do princípio anterior
 - Desejo x Necessidade
 - Liberdade para alterações futuras

Tipos Abstratos de Dados (TADs)

Níveis de detalhamento



Implementação de TADs

- Em linguagens estruturadas (C, Pascal), a implementação é feita com definições de tipos e implementação de funções
 - *typedef*
- Em linguagens orientadas a objetos a implementação é feita através de classes (começaremos usando structs!)
 - C++, Java, C#, ...

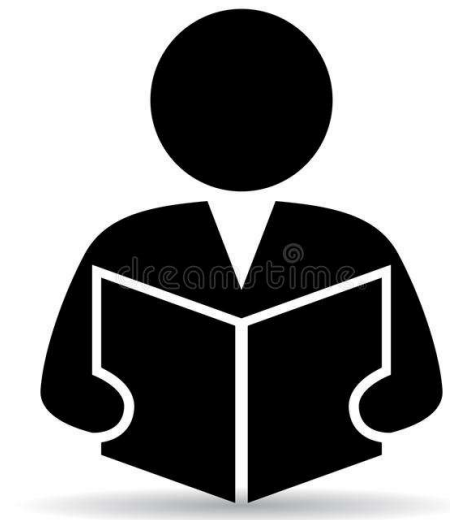
Structs

- Em **C++** uma **struct** é similar a uma **classe**
 - Diferença no nível de proteção (outra aula!)
- Conjunto de variáveis / atributos (possivelmente de tipos diferentes) e operações / métodos agrupados e acessíveis sob um único nome (significado)
- Exemplos:
 - Aluno
 - Ponto 3D

Structs

Exemplo 1

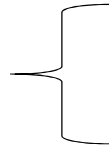
- Como representar um Aluno?
 - Quais atributos/dados?
 - Nome, matrícula, curso, ...
 - Quais operações sobre esses dados?
 - Matricular, Calcular RSG, ...



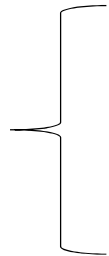
Structs

Exemplo 1

Dados/Atributos



Operações /
Métodos



```
#include <iostream>
#include <string>
using namespace std;

struct Aluno {
    string nome;
    int matricula;

    float calcularRSG() {
        // Fazer a conta necessaria
        return 0;
    }
};
```

← Atenção ao 'ponto e vírgula' na definição!

Structs

Exemplo 1

Acessando os dados

(aqui estamos acessando dados diretamente, mas uma boa técnica de programação é ter métodos para isso)

```
int main() {  
    Aluno a1;
```

Declarando uma variável do
tipo Aluno
(alocação estática -> pilha)

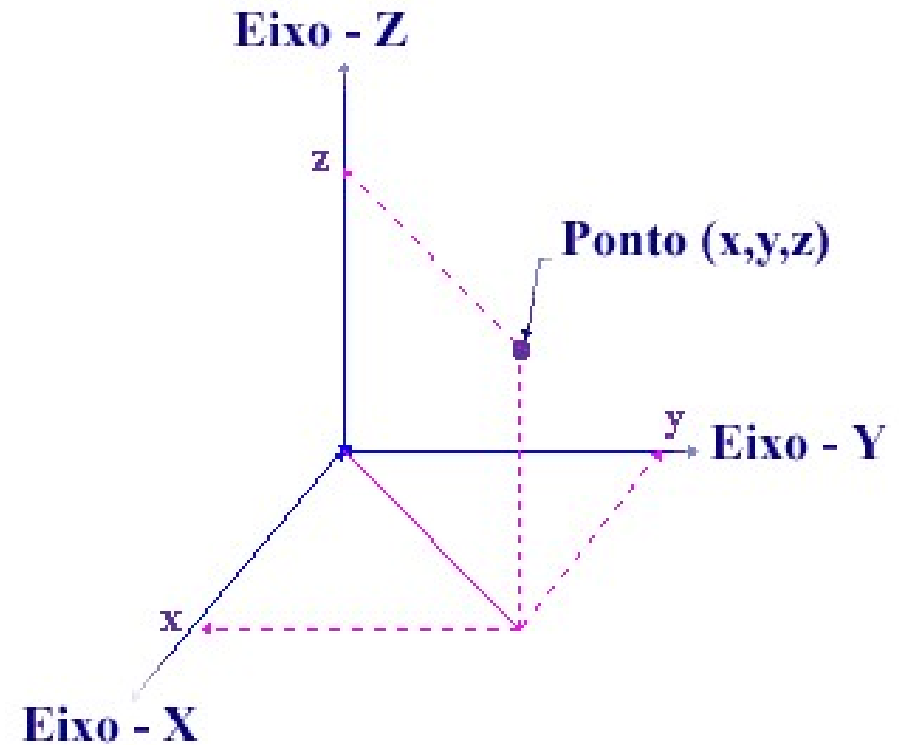
```
    {  
        a1.nome = "Jose da Silva";  
        a1.matricula = 201812345;  
        cout << a1.nome << endl;  
        cout << a1.calcularRSG();  
        return 0;  
    }
```

Chamando um método.

[Code](#)

Structs

- Ponto 3D
 - Atributos: x, y, z
 - Métodos: *CalculaDistância*



Structs

Exemplo 2

Dados/Atributos

Operações
/ Métodos

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

Funções matemáticas

```
using namespace std;
```

```
struct Ponto3D {
    double x;
    double y;
    double z;
```

Ponteiro para a **própria** estrutura. Não é obrigatório aqui, mas facilita o entendimento. (Será melhor discutido no futuro).

```
double calcularDistancia(Ponto3D* p2) {
    double dx = p2->x - this->x;
    double dy = p2->y - this->y;
    double dz = p2->z - this->z;
    return sqrt(dx*dx + dy*dy + dz*dz);
}
};
```

<https://www.cplusplus.com/reference/cmath/>

Structs

Exemplo 2

```
int main() {  
  
    Ponto3D* p1 = new Ponto3D();  
    p1->x = 0.0;  
    p1->y = 0.0;  
    p1->z = 0.0;  
  
    Ponto3D* p2 = new Ponto3D();  
    p2->x = 5.0;  
    p2->y = 5.0;  
    p2->z = 5.0;  
  
    cout << p1->calcularDistancia(p2) << endl;  
  
    delete p1;  
    delete p2;  
  
    return 0;  
}
```

Alocando as diferentes variáveis na memória.

Chamando um método.

← **Atenção!**

Structs

Construtores

- Tipo especial de membro do TAD
 - Implícitos (default) / Explícitos
- Ajudam na inicialização dos elementos
- São chamados na declaração/
alocação

```
int main() {  
  
    Ponto3D* p = new Ponto3D();  
    p->Imprime();  
  
    return 0;  
}
```

-1 -1 -1

```
struct Ponto3D {  
    double x;  
    double y;  
    double z;  
  
    Ponto3D() {  
        this->x = -1;  
        this->y = -1;  
        this->z = -1;  
    }  
  
    Ponto3D(double x, double y, double z) {  
        this->x = x;  
        this->y = y;  
        this->z = z;  
    }  
  
    void Imprime() {  
        cout << x << " " << y << " " << z;  
    }  
};
```

[Code](#)

Structs

Construtores

- O que vai ser impresso?

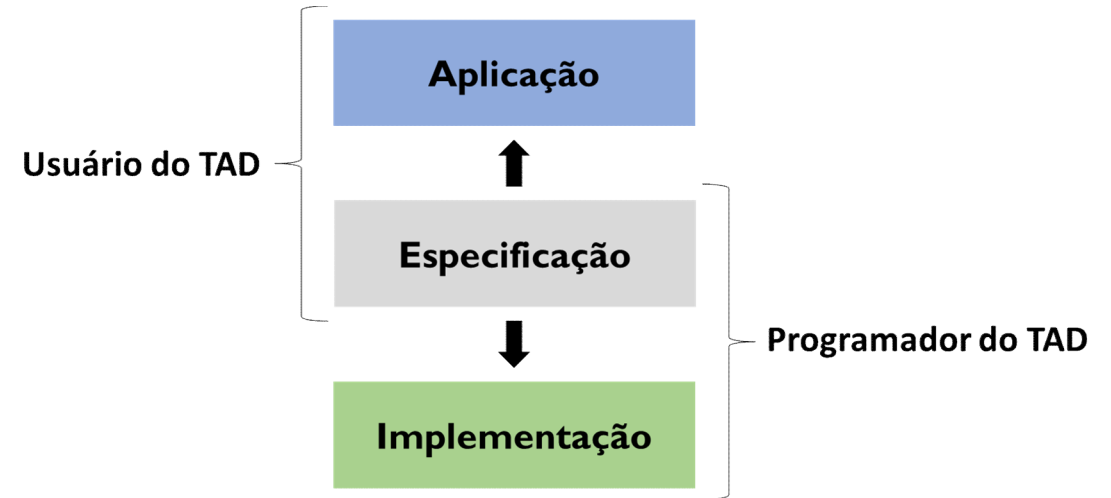
```
luizch@DESKTOP-URUBBRT:  
Global: 0  
Local: 0  
Ponto: 0 0 0
```

```
luizch@Dell2023:/mnt/c/Users/l  
Global: 0  
Local: -432591224  
Ponto: 32686 -432592576 32686
```

```
#include <iostream>  
using namespace std;  
  
int g;  
  
struct Ponto3D {  
    int x, y, z;  
  
    void Imprime() {  
        cout << "Ponto: " << x << " "  
             << y << " " << z << endl;  
    }  
};  
  
int main() {  
    Ponto3D p1;  
    int x;  
  
    cout << "Global: " << g << endl;  
    cout << "Local: " << x << endl;  
    p1.Imprime();  
}
```

Structs

- Os exemplos seguem corretamente todos os conceitos de TADs que falamos anteriormente?
- Qual o problema?
 - **Especificação** e **Implementação** estão **juntas**!
- Como resolver isso?
 - Separar → Modularizar

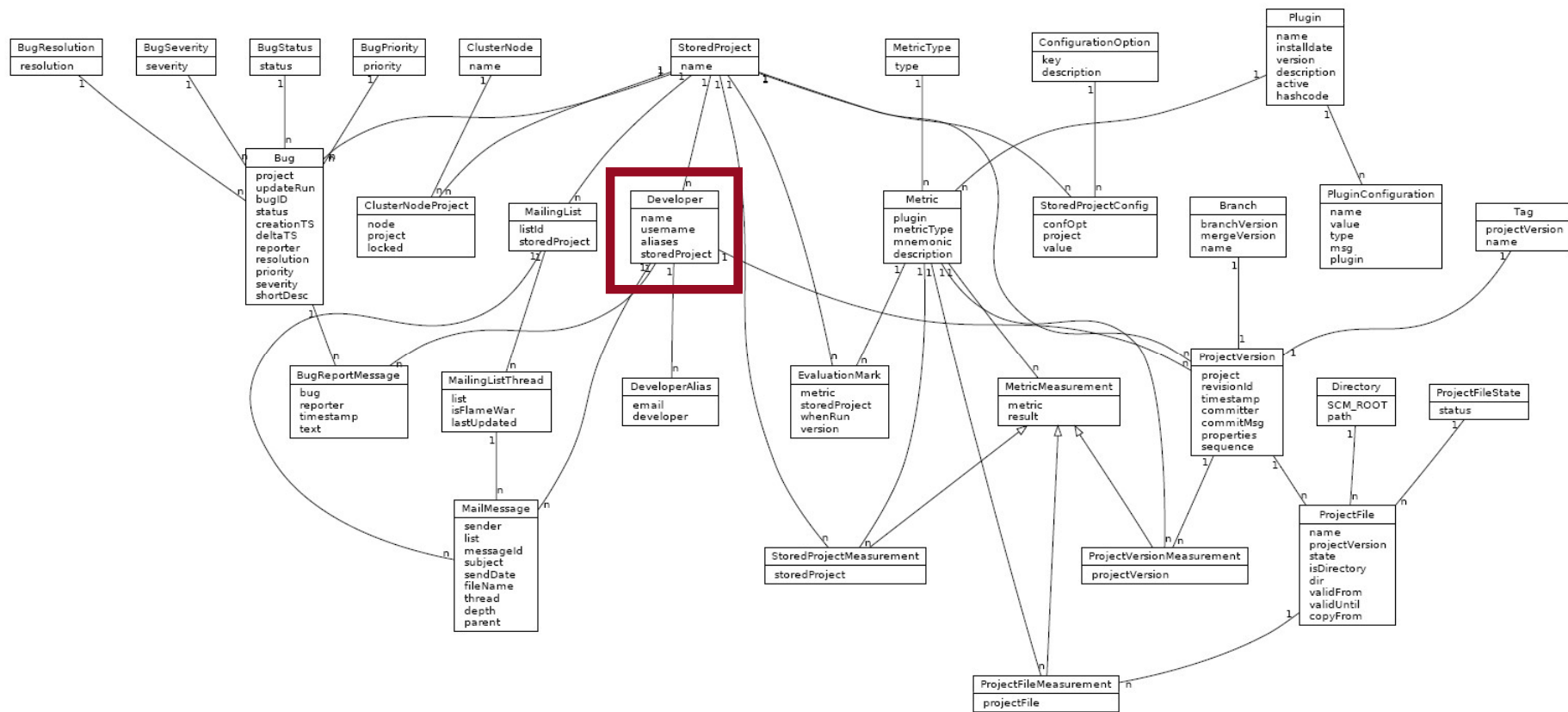


```
struct Aluno {  
    string nome;  
    int matricula;  
  
    float calcularRSG() {  
        // Fazer a conta necessaria  
        return 0;  
    }  
};
```

Modularização

À medida que a complexidade dos programas aumenta, é inviável manter tudo em um mesmo arquivo / módulo.

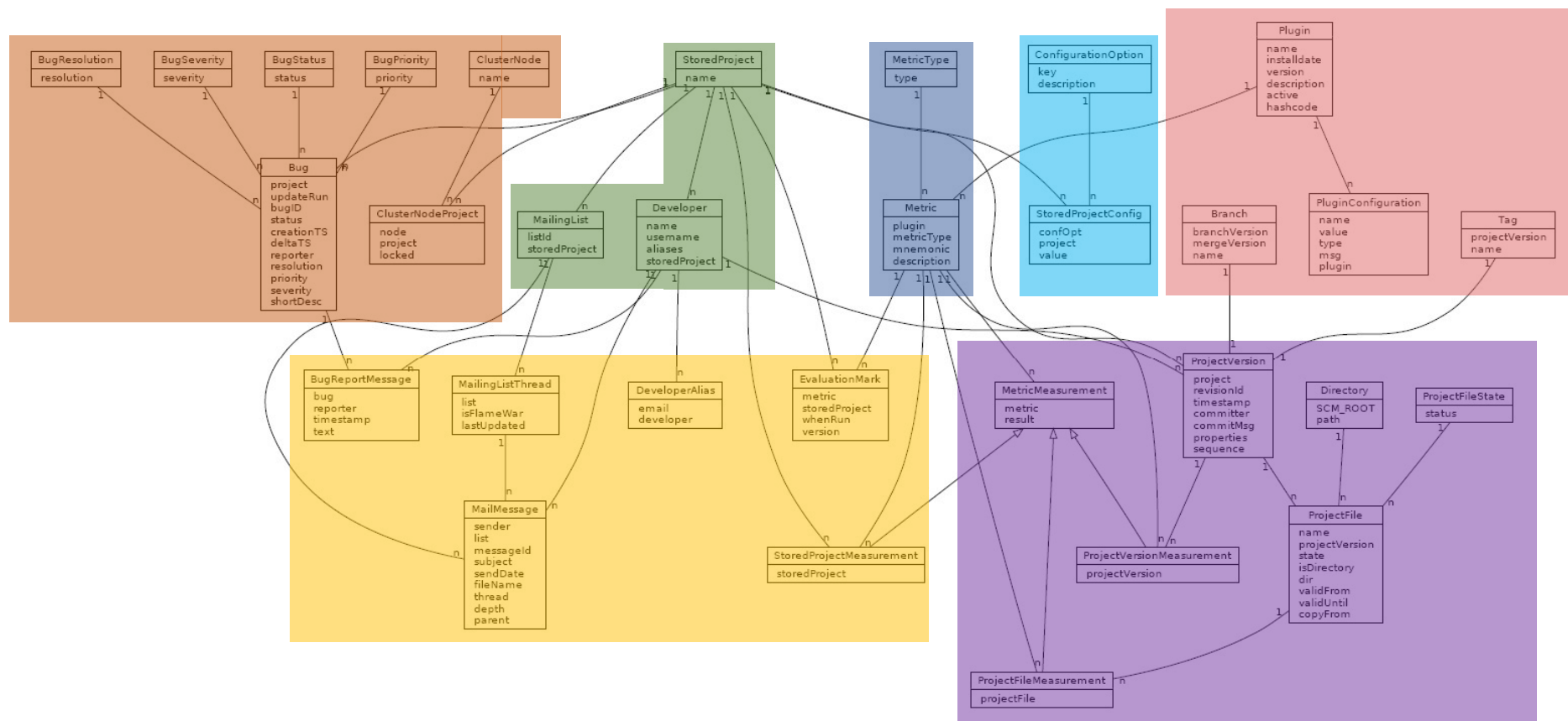
Modularização



Modularização

- Programação modular
 - Separar aspectos da funcionalidade do programa
 - Partes independentes e intercambiáveis
 - “Contrato”
 - Modificar, compilar e (re)utilizar individualmente

Modularização

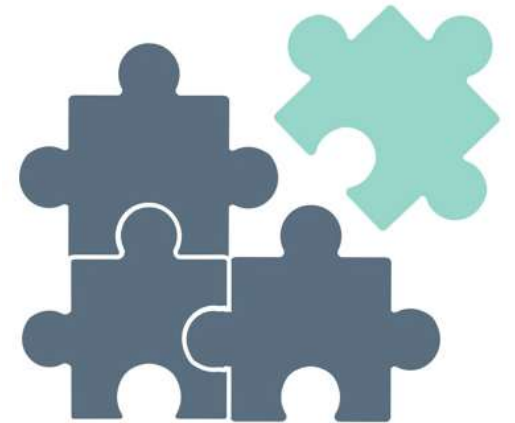


Modularização

- Abstração e Encapsulamento
- Permite uma melhor organização do código
- Facilita o trabalho em equipe
- Facilita a Manutenção e Testes
- Permite o Reuso

Módulo

- Propósito único
- Interface apropriada com outros módulos
- Pode ser compilado separadamente
- Reutilizáveis e modificáveis



Coesão e Acoplamento

- Coesão

- Grau de **intradependência** entre os elementos do módulo
- Funções, responsabilidades (mesmo objetivo)

- Acoplamento

- Grau de **interdependência** entre diferentes módulos
- Alteração em um demanda alteração no outro

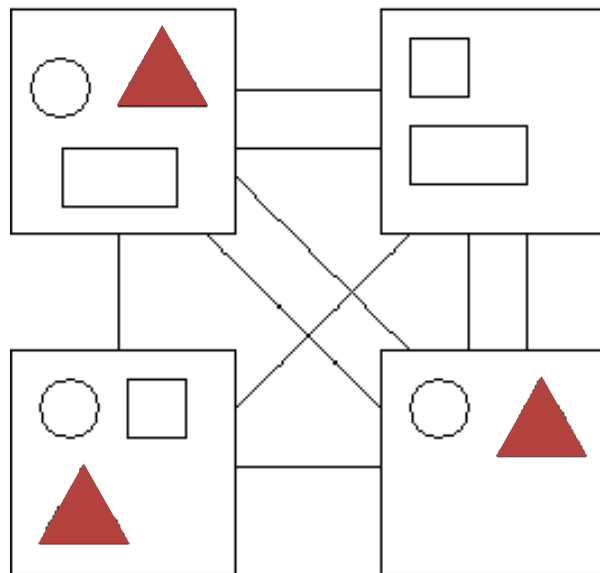
Coesão e Acoplamento

 **Coesão**

Acoplamento 

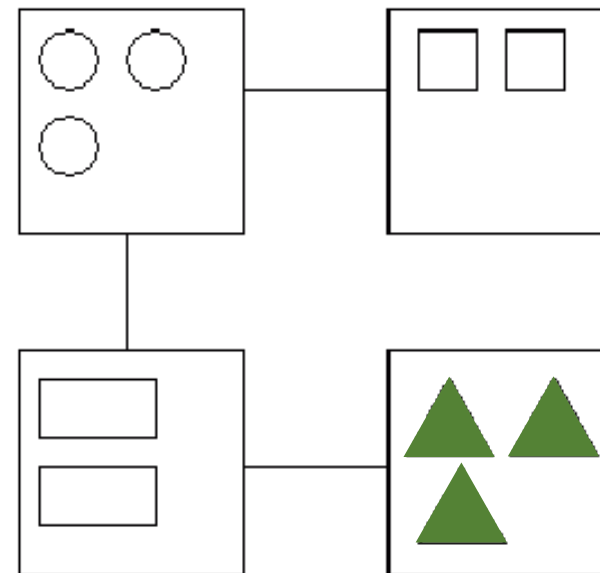
Coesão e Acoplamento

SISTEMA 1



Baixa Coesão / Alto Acoplamento

SISTEMA 2



Alta Coesão / Baixo Acoplamento

Modularização em C++

Principais Recursos:

- Uso de TADs (structs e classes)
- Separação em arquivos, compilação em separado
- Namespaces
- Modules (a partir do C++ 20)

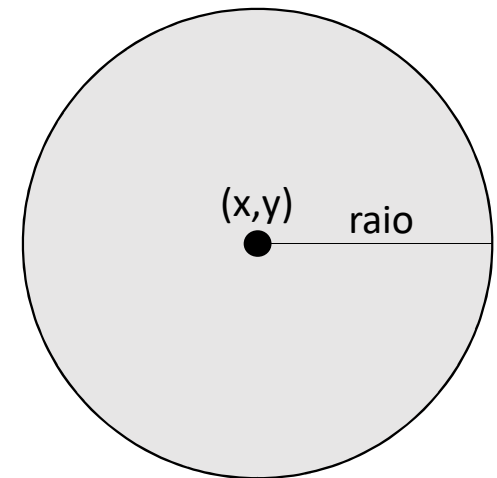
Modularização

- Separação em arquivos
 - **Especificação**: NomeDoTAD.**hpp** (arquivo de cabeçalho)
 - Define o que chamamos de **contrato** do TAD
 - **Implementação**: NomeDoTAD.**cpp**
- Para se utilizar esse TAD no main ou em outras partes do programa é necessário acessar via **#include** do arquivo **.hpp**
 - **Não** deve-se fazer include do arquivo **.cpp**!

Modularização

Exemplo 3

- Como representar uma Circunferência?
 - Quais atributos/dados?
 - Coordenadas do centro, raio, ...
 - Quais operações sobre esses dados?
 - Calcular Área, Perímetro, ...



Modularização

Exemplo 3 Circunferencia.hpp (arquivo de cabeçalho / header)

Guarda:

Evita
múltiplas
declarações

https://en.wikipedia.org/wiki/Include_guard

```
#ifndef CIRCUNFERENCIA_H  
#define CIRCUNFERENCIA_H
```

```
struct Circunferencia {  
    double _x, _y;  
    double _raio;  
};
```

```
Circunferencia(double, double, double);  
double calcularArea();
```

```
};
```

```
#endif
```

Na declaração, não é obrigatório informar o nome dos argumentos, mas é uma boa prática.

O contrato (.hpp) possui apenas o cabeçalho das funções.

Modularização

Exemplo 3

Circunferencia.cpp (arquivo de implementação do TAD)

Fazer o include do arquivo de cabeçalho (**contrato**) do TAD que será implementado.

```
#include "Circunferencia.hpp"
#include <cmath>
Circunferencia::Circunferencia(double x, double y, double raio)
{
    _x = x;
    _y = y;
    _raio = raio;
}

double Circunferencia::calcularArea() {
    return M_PI * pow(_raio, 2);
}
```

A qual TAD se refere.

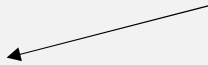
Operação sendo implementada.

Modularização

Exemplo 4 `main.cpp`

```
#include <iostream>
#include "Circunferencia.hpp"
```

Só precisa incluir o **contrato**
para utilizar o TAD!



```
using namespace std;
```

```
int main() {
```

```
    Circunferencia* circ = new Circunferencia(0, 0, 10);
    cout << circ->calcularArea() << endl;
```

```
    delete circ;
```

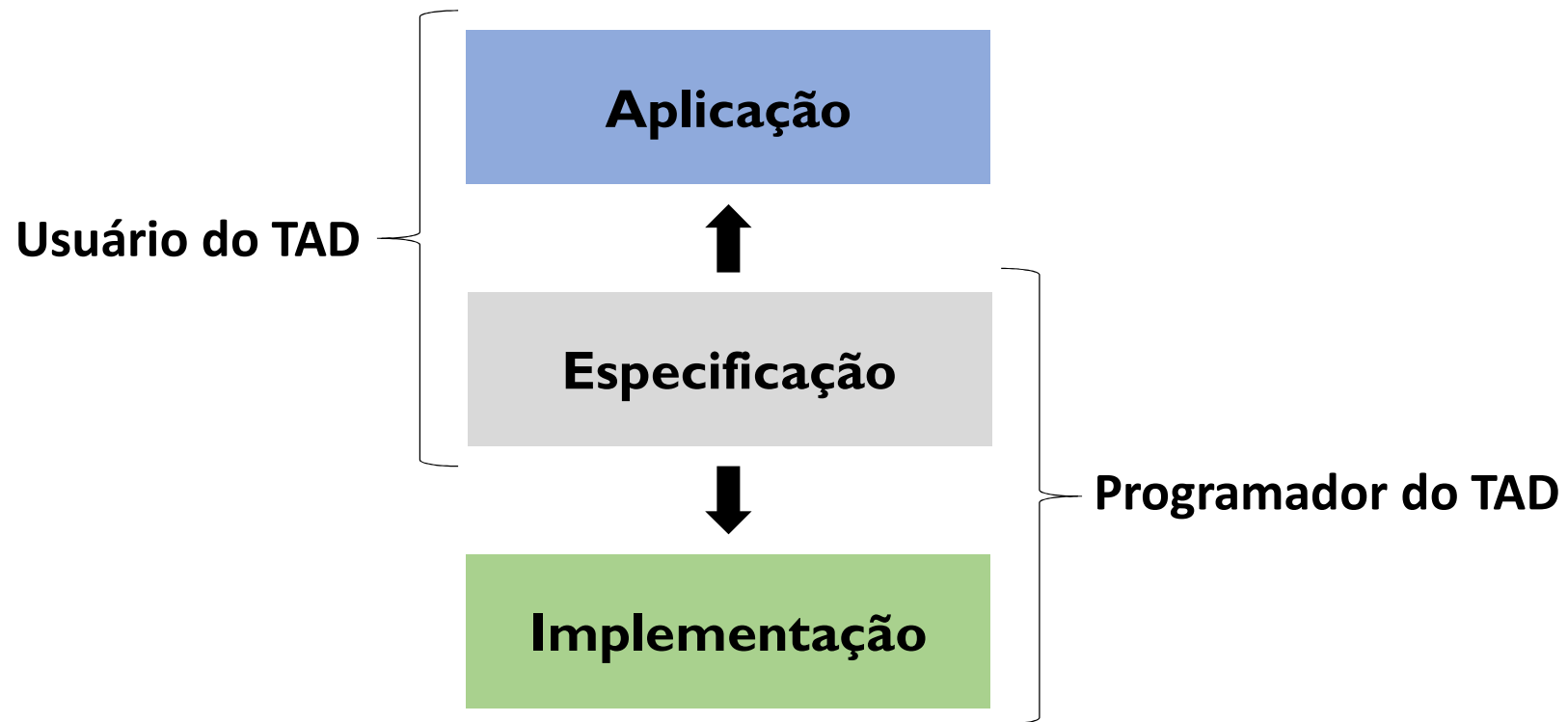
```
    return 0;
```

```
}
```

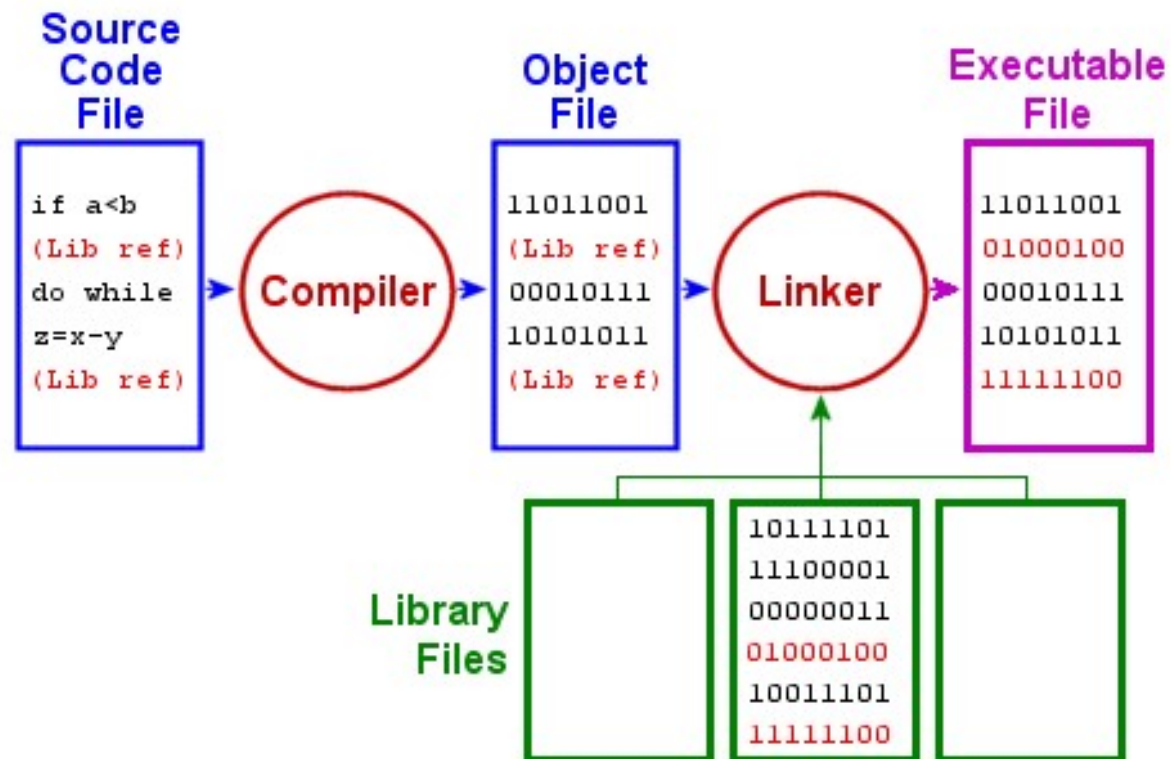
Compilação

- Grandes sistemas
 - Equipes de programadores trabalhando em paralelo
 - Código distribuído em vários arquivos fonte
- Não é conveniente (tempo, recursos) recompilar partes do programa que não foram alteradas (ou ele por inteiro)
- Princípio do Encapsulamento
 - Separar a especificação (contrato) de como o TAD é usado, dos detalhes específicos da sua implementação interna

Compilação



Compilação



Compilação

■ Compilar e Ligar

```
g++ arquivo1.cpp arquivo2.cpp -o Executavel
```

■ Compilar apenas

```
g++ -c arquivo.cpp -o arquivo.o
```

■ Ligar apenas

```
g++ arquivo1.o arquivo2.o -o Executavel
```

Compilação - Exemplo

Compilar e linkar de forma conjunta

```
g++ main.cpp Circunferencia.cpp -o main
```

- Main.cpp e Circunferencia.cpp tem são compilados juntos e “linkados” no arquivo executável main
- Problemas:
 - Necessário o acesso ao código fonte do TAD circunferência
 - Arquivos são compilados mesmo sem ter sido modificados

Compilação - Exemplo

Compilar separadamente e linkar depois

```
g++ -c Circunferencia.cpp -o circunferência.o  
g++ -c main.cpp -o main.o  
g++ main.o Circunferencia.o -o main
```

- Vantagens:
 - O usuário do TAD não precisa do .cpp
 - Compilar somente o que é necessário
- Dificuldade: como controlar esse processo?
 - **Makefiles!**

Compilação

Makefile

- Funciona como um roteiro para a compilação
 - Arquivo de texto especialmente formatado (tabs)
 - Entrada para um utilitário Unix chamado 'make'
- Contém uma lista de requisitos para que um programa seja considerado 'up to date' (versão mais recente do código)
 - O utilitário *make* examina esses requisitos, verifica os *timestamps* em todos os arquivos listados no makefile e recompila apenas os arquivos com um registro desatualizado

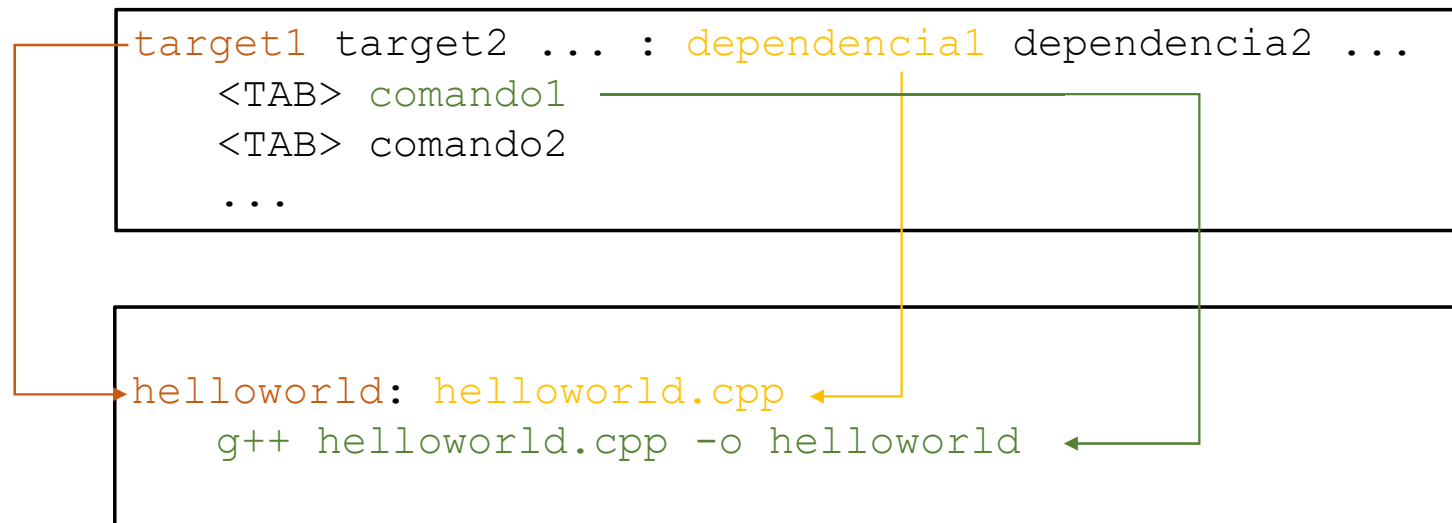
<https://www.gnu.org/software/make/manual/make.html>

<https://www.cs.bu.edu/teaching/cpp/writing-makefiles/>

Compilação

Makefile

- Makefile contém atribuições de variáveis, comentários e regras (targets)
 - Informa as dependências entre os arquivos e targets
 - Indica os comandos necessários para a compilação



Compilação

Makefile

```
CC=g++
```

```
CFLAGS=-std=c++11 -Wall
```

← Variáveis auxiliares

```
all: main
```

```
circunferencia.o: circunferencia.hpp circunferencia.cpp  
    ${CC} ${CFLAGS} -c circunferencia.cpp
```

```
main.o: circunferencia.hpp main.cpp  
    ${CC} ${CFLAGS} -c main.cpp
```

```
main: main.o circunferencia.o  
    ${CC} ${CFLAGS} main.o circunferencia.o -o main
```

```
# Rule for cleaning files generated during compilation.  
# Call 'make clean' to use it  
clean:
```

```
    rm -f main *.o
```

Dependências

Targets

(primeiro é
o default)

Compilação

Makefile

```
> make
g++ -std=c++11 -Wall -c main.cpp
g++ -std=c++11 -Wall -c circunferencia.cpp
g++ -std=c++11 -Wall main.o circunferencia.o -o main
> ./main
> make
g++ -std=c++11 -Wall -c circunferencia.cpp
g++ -std=c++11 -Wall main.o circunferencia.o -o main
> ./main
> make
make: Nothing to be done for 'all'.
> make clean
rm -f main *.o
```

(Após alguma alteração apenas em circunferencia.cpp)

**O arquivo main.cpp
não precisou ser
recompilado!**

Compilação

Makefile

Cuidado!

O makefile depende de timestamps, portanto verifique se a data / hora do sistema está de acordo

- “Clockskew”
- WSL pode estar diferente do Windows: `sudo hwclock -s`

Organização

- Separação em diferentes diretórios
 - Agrupamento físico de uma estrutura lógica (associação)

```
. project
├── Makefile
├── build
│   └── [objects]
├── include
│   ├── modulo1
│   │   ├── mod1c1.hpp
│   │   └── mod1c2.hpp
├── src
│   ├── main.cpp
│   ├── modulo1
│   │   ├── mod1c1.cpp
│   │   └── mod1c2.cpp
└── test
```

Colocar em diferentes diretórios:

← Arquivos compilados;

← Arquivos de especificação;

← Arquivos de implementação.

Compilação

Makefile

```
CC=g++
CFLAGS=-std=c++11 -Wall
SRC_DIR=src
INCLUDE_DIR=include
OBJ_DIR=obj

all: main

$(OBJ_DIR)/circunferencia.o: $(INCLUDE_DIR)/circunferencia.hpp $(SRC_DIR)/circunferencia.cpp
    $(CC) $(CFLAGS) -c $(SRC_DIR)/circunferencia.cpp -I$(INCLUDE_DIR) -o
$(OBJ_DIR)/circunferencia.o

$(OBJ_DIR)/main.o: $(INCLUDE_DIR)/circunferencia.hpp $(SRC_DIR)/main.cpp
    $(CC) $(CFLAGS) -c $(SRC_DIR)/main.cpp -I$(INCLUDE_DIR) -o $(OBJ_DIR)/main.o

main: $(OBJ_DIR)/main.o $(OBJ_DIR)/circunferencia.o
    $(CC) $(CFLAGS) $(OBJ_DIR)/main.o $(OBJ_DIR)/circunferencia.o -o main

clean:
    rm -f main $(OBJ_DIR)/*.o
```

Variáveis com os diretórios

Flag de compilação para Indicar onde estão os .hpp