Projeto Final para Disciplina de Programação Avançada

Sistema de Gestão Esportiva

Uma Abordagem Orientada a Objetos para Gerenciamento de Torneios

Autor

Thiago dos Santos*

Disciplina: Programação Avançada - Prof. Dr. Robson R. Linhares

Departamento: PPGCA | Departamento Acadêmico de Informática – DAINF - Campus Curitiba

Universidade: Universidade Tecnologia Federal Paraná (UTFPR)

27 de novembro de 2025

Resumo

Este trabalho apresenta o desenvolvimento de um Sistema de Gestão Esportiva implementado em C++ como projeto final da disciplina de Programação Avançada. O sistema foi desenvolvido utilizando princípios robustos de Programação Orientada a Objetos (POO) para gerenciar torneios esportivos, equipes, jogadores e rankings de forma modular e extensível. A metodologia de desenvolvimento incluiu elicitação de requisitos, modelagem com diagramas UML, e implementação em C++ com foco em boas práticas de design. O sistema demonstra a aplicação prática de conceitos avançados de POO, incluindo encapsulamento, herança, polimorfismo, classes abstratas, templates C++, e uso extensivo da Standard Template Library (STL). A persistência de dados foi implementada através de arquivos CSV, garantindo portabilidade e simplicidade. O projeto resultou em um sistema funcional, testado e documentado, capaz de gerenciar competições esportivas em duas modalidades (pontos corridos e mata-mata) com geração automática de rodadas e rankings dinâmicos.

Palavras-chave: Programação Orientada a Objetos, C++, Gestão Esportiva, Arquitetura de Software, Padrões de Projeto.

*email: thiagodossantos315@gmail.com

Resumo

This paper presents the development of a Sports Management System implemented in C++ as a final project for the Advanced Programming course. The system was developed using robust Object-Oriented Programming (OOP) principles to manage sports tournaments, teams, players, and rankings in a modular and extensible manner. The development methodology included requirements elicitation, modeling with UML diagrams, and C++ implementation focusing on design best practices. The system demonstrates practical application of advanced OOP concepts, including encapsulation, inheritance, polymorphism, abstract classes, C++ templates, and extensive use of the Standard Template Library (STL). Data persistence was implemented through CSV files, ensuring portability and simplicity. The project resulted in a functional, tested, and documented system capable of managing sports competitions in two modalities (round-robin and knockout) with automatic round generation and dynamic rankings.

Keywords: Object-Oriented Programming, C++, Sports Management, Software Architecture, Design Patterns.

1 INTRODUÇÃO

1.1 Contexto

A disciplina de Programação Avançada tem como objetivo capacitar os alunos no desenvolvimento de aplicações de médio a grande porte utilizando princípios sólidos de Engenharia de Software e Programação Orientada a Objetos. A escolha pela linguagem C++ se justifica pelo fato de ser uma linguagem compilada, com forte tipagem estática, suporte robusto a POO, e amplo ecossistema de bibliotecas padrão.

1.2 Objetivo

O objetivo geral deste projeto é desenvolver um sistema completo de gerenciamento de torneios esportivos que demonstre o domínio de conceitos avançados de OOP e boas práticas de desenvolvimento de software. Especificamente, buscamos:

1. Aplicar princípios fundamentais de POO (encapsulamento, herança, polimorfismo)
2. Projetar e implementar uma arquitetura modular e escalável
3. Implementar persistência de dados robusta
4. Aplicar padrões de projeto reconhecidos
5. Desenvolver código limpo, bem documentado e facilmente mantível
6. Validar a solução através de testes e exemplos de uso

1.3 Objeto de Estudo

O objeto de estudo é um **Sistema de Gestão de Torneios Esportivos** que encapsula toda a lógica necessária para gerenciar:

- Cadastro e manutenção de jogadores com atributos heterogêneos
- Formação de equipes com múltiplos jogadores
- Criação e configuração de torneios em múltiplas modalidades
- Geração automática de rodadas e tabelas de confrontos
- Simulação de partidas e registro de resultados
- Cálculo e atualização de rankings em tempo real
- Persistência e recuperação de dados entre execuções

1.4 Metodologia

A metodologia de desenvolvimento compreendeu etapas de elicitação de requisitos e levantamento de funcionalidades essenciais, seguidas pela análise e modelagem com uso de diagramas UML (incluindo diagramas de classes, sequência e casos de uso). Em seguida, procedeu-se ao design da arquitetura com definição da estrutura em camadas e aplicação de padrões de projeto reconhecidos. A implementação do sistema foi realizada em C++ conforme os padrões estabelecidos, e as funcionalidades foram verificadas por meio de testes e validação de casos de uso. Por fim, toda a produção do projeto foi registrada com documentação técnica apropriada e elaboração do relatório final.

2 EXPLICAÇÃO DO SOFTWARE

2.1 Funcionalidades Principais do Ponto de Vista do Usuário

O sistema apresenta uma interface de linha de comando que oferece ao usuário as seguintes funcionalidades:

1. **Gerenciamento de Jogadores**
    - Cadastrar novos jogadores com nome, idade, esporte e tipo (amador/profissional)
    - Listar todos os jogadores cadastrados
    - Visualizar estatísticas de um jogador (vitórias, derrotas, empates)
    - Remover jogadores do sistema
2. **Gerenciamento de Equipes**

- Criar novas equipes
- Adicionar/remover jogadores a uma equipe
- Listar equipes e seus membros
- Visualizar estatísticas agregadas de uma equipe

3. **Criação e Gerenciamento de Torneios**
   - Criar torneio escolhendo modalidade (pontos corridos ou mata-mata)
   - Adicionar equipes/jogadores ao torneio
   - Gerar rodadas automaticamente
   - Visualizar a tabela de jogos
4. **Simulação de Partidas**
   - Simular resultado de uma partida
   - Registrar resultados reais (vitória, derrota, empate)
   - Atualizar pontuações e estatísticas dos participantes
5. **Gerenciamento de Rankings**
   - Gerar ranking de participantes
   - Ordenar por critérios (pontos, vitórias, saldo)
   - Exportar ranking em arquivo CSV
   - Visualizar histórico de rankings
6. **Persistência de Dados**
   - Salvar automaticamente dados ao encerrar
   - Carregar dados ao iniciar
   - Importar dados de arquivos CSV
   - Garantir integridade dos dados

2.2 Fluxo de Interação Típico

Um fluxo típico de interação com o sistema segue os seguintes passos:

1. Usuário inicia a aplicação
2. Sistema carrega dados persistidos de rodadas anteriores
3. Usuário visualiza menu principal com opções

4. Usuário cadastra jogadores e forma equipes
5. Usuário cria um novo torneio escolhendo a modalidade
6. Sistema gera rodadas automaticamente
7. Usuário simula/registra resultados das partidas
8. Sistema atualiza ranking em tempo real
9. Usuário visualiza ranking atualizado
10. Usuário exporta dados para análise externa
11. Usuário encerra a aplicação
12. Sistema salva todos os dados para próxima sessão

3 DESENVOLVIMENTO DO SOFTWARE NA VERSÃO ORIENTADA A OBJETOS

3.1 Requisitos Funcionais

A Tabela 1 lista os requisitos detalhados do sistema propostos para o projeto:

ID	Requisito	Peso
1	Permitir cadastro de jogadores com atributos básicos (nome, idade, esporte)	10
2	Permitir cadastro de equipes compostas por jogadores	10
3	Criar torneios individuais ou coletivos	10
4	Gerar automaticamente rodadas do torneio	10
5	Registrar resultados de partidas manualmente	5
6	Simular resultados automaticamente	5
7	Gerar ranking de jogadores/equipes	5
8	Persistir e recuperar dados em arquivo texto e/ou binário utilizando padrão de persistência DAO	10
9	Recuperar dados salvos para continuar torneio	0
10	Exibir estatísticas de desempenho (vitórias, derrotas, empates)	5
11	Permitir diferentes modalidades de torneio (ex.: mata-mata, pontos corridos)	5
12	Interface textual simples para interação do usuário	0
13	Suporte a diferentes tipos de jogadores (ex.: amador, profissional)	5
14	Possibilidade de edição de dados de jogadores/equipes	10
15	Exportar ranking final em arquivo separado utilizando padrão de persistência DAO	10

Tabela 1: Requisitos detalhados do sistema e respectivos pesos

A Tabela 2 apresenta os requisitos funcionais do sistema com seus pesos, prioridades e status de implementação:

ID	Requisito	Peso	Prioridade	Status
1	Permitir cadastro de jogadores com atributos básicos (nome, idade, esporte)	10	ALTA	✓
2	Permitir cadastro de equipes compostas por jogadores	10	ALTA	✓
3	Criar torneios individuais ou coletivos	10	ALTA	✓
4	Gerar automaticamente rodadas do torneio	10	ALTA	✓
5	Registrar resultados de partidas manualmente	5	MÉDIA	✓
6	Simular resultados automaticamente	5	MÉDIA	✓
7	Gerar ranking de jogadores/equipes	5	ALTA	✓
8	Persistir e recuperar dados em arquivo texto e/ou binário utilizando padrão de persistência DAO	10	ALTA	✓
9	Recuperar dados salvos para continuar torneio	0	BAIXA	✓
10	Exibir estatísticas de desempenho (vitórias, derrotas, empates)	5	MÉDIA	✓
11	Permitir diferentes modalidades de torneio (ex.: mata-mata, pontos corridos)	5	ALTA	✓
12	Interface textual simples para interação do usuário	0	ALTA	✓
13	Suporte a diferentes tipos de jogadores (ex.: amador, profissional)	5	ALTA	✓
14	Possibilidade de edição de dados de jogadores/equipes	10	MÉDIA	✓
15	Exportar ranking final em arquivo separado utilizando padrão de persistência DAO	10	ALTA	✓

Tabela 2: Requisitos Funcionais do Sistema (atualizado)

3.2 Diagrama de Classes

O diagrama de classes (Figura 1) mostra a estrutura estática do sistema com as sete classes principais e seus relacionamentos:

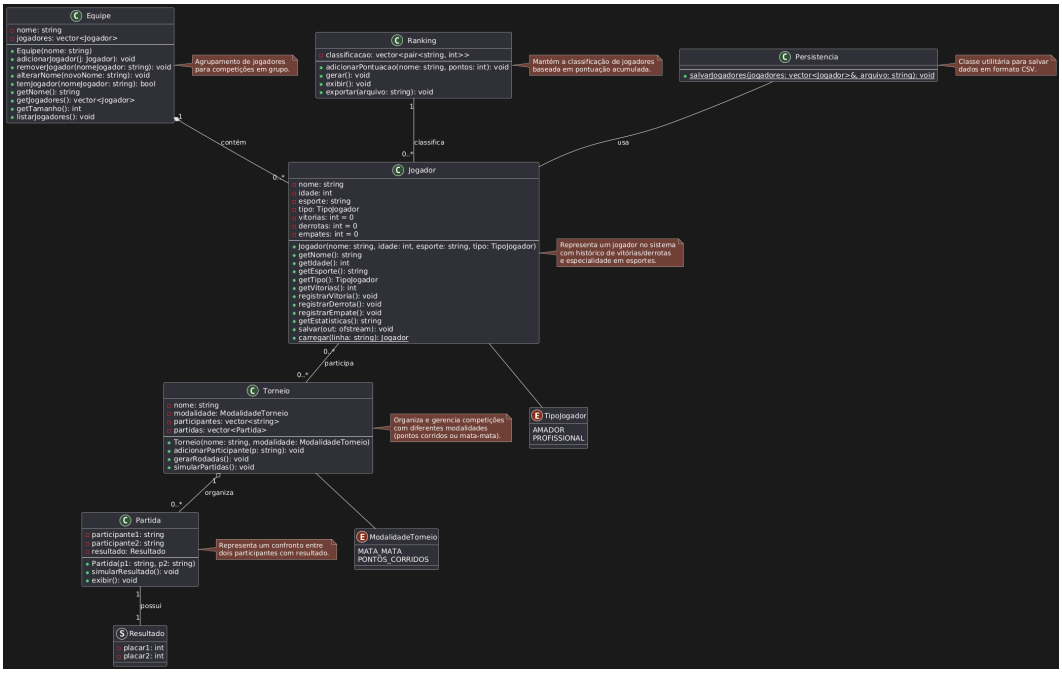


Figura 1: Diagrama UML de Classes do Sistema de Gestão Esportiva. A figura mostra a estrutura estática com as classes Jogador, Equipe, Partida, Torneio, Ranking, Persistencia e Enums, incluindo seus atributos, métodos e relacionamentos (composição, agregação e dependência).

3.3 Estrutura de Classes

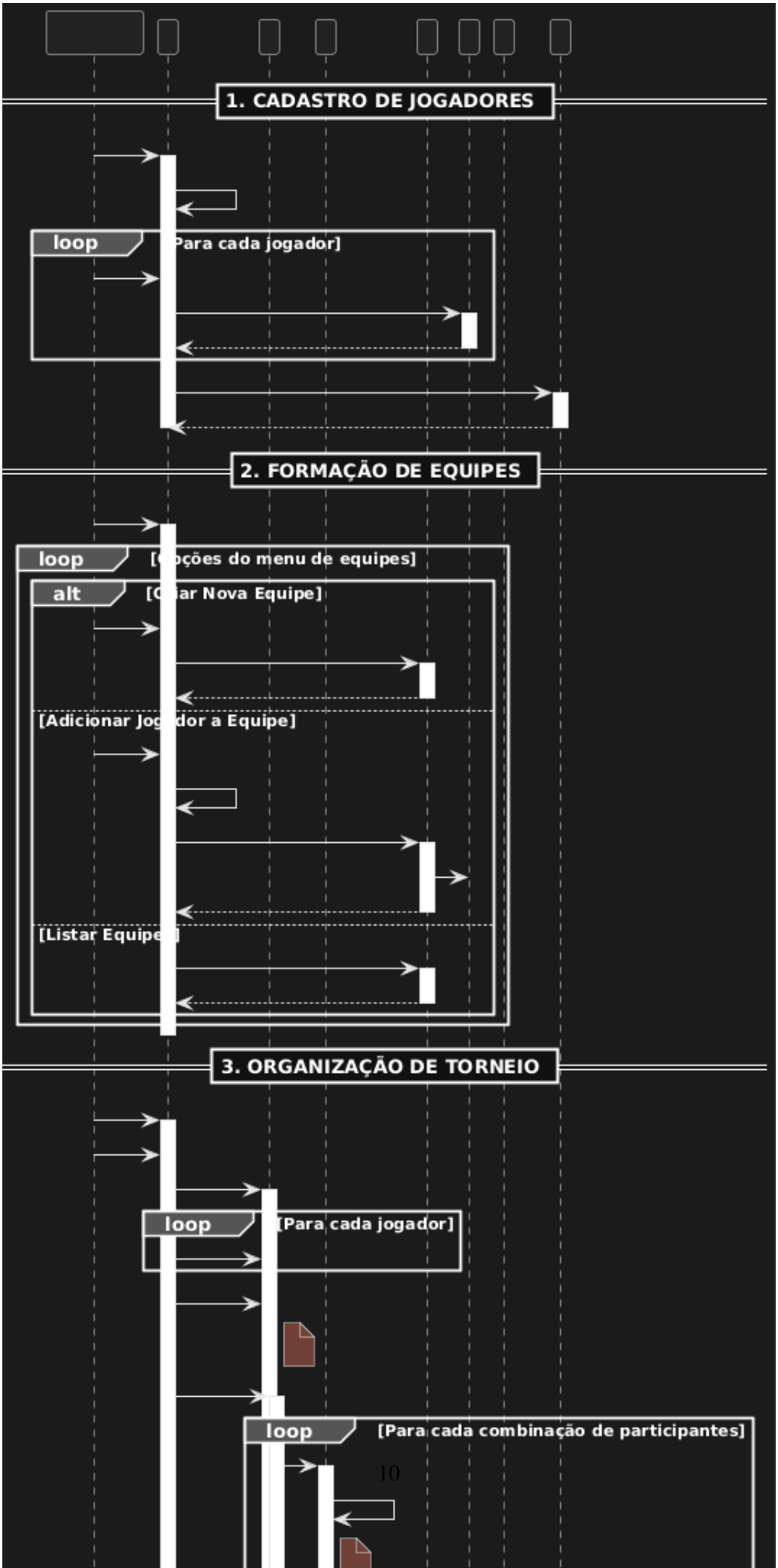
A Tabela 3 apresenta um resumo das sete classes principais:

Classe	**Responsabilidade Principal**	**Relacionamentos**
`Jogador`	Encapsular dados e estatísticas de um jogador individual	-
`Equipe`	Agregar múltiplos jogadores sob uma identidade comum	1..N `Jogador`
`Partida`	Registrar confronto entre dois participantes	2 `Equipe/Jogador`
`Torneio`	Orquestrar competição completa	N `Partida`, N `Equipe`
`Ranking`	Classificar participantes por critérios	N `Jogador/Equipe`
`Persistencia`	Abstrair operações de I/O e persistência	Todas
`Enums`	Definir constantes e tipos enumerados	Todas

Tabela 3: Estrutura de Classes do Sistema

3.3.1 Diagrama de Sequência para Criação de Torneio

O diagrama de sequência (Figura 2) ilustra a interação entre objetos quando um torneio é criado e iniciado:


1. CADASTRO DE JOGADORES
loop
Para cada jogador]
2. FORMAÇÃO DE EQUIPES
loop
alt
[Adicionar Jog dor a Equipe]
[Listar Equipe
3. ORGANIZAÇÃO DE TORNEIO
loop
[Para cada jogador]
loop
[Para cada combinação de participantes]

3.4 Descrição Detalhada das Classes

3.4.1 Classe Jogador

A classe `Jogador` representa a unidade atômica do sistema, encapsulando dados pessoais e estatísticas de desempenho.

```
class Jogador {
private:
    string nome;
    int idade;
    string esporte;
    TipoJogador tipo;  // AMADOR ou PROFISSIONAL
    int vitorias;
    int derrotas;
    int empates;

public:
    Jogador(string n, int id, string esp, TipoJogador t);


    string getNome() const;
    int getIdade() const;
    string getEsporte() const;
    TipoJogador getTipo() const;

    void registrarVitoria();
    void registrarDerrota();
    void registrarEmpate();
    int getTotalPartidas() const;
    double getTaxaSucesso() const;
};
```

Listing 1: Definição da classe Jogador

Atributos principais:

- `nome`: Identificador único do jogador
- `idade`: Idade em anos (inteiro)
- `esporte`: Modalidade esportiva (ex: Futebol, Vôlei)
- `tipo`: Classificação do jogador (amador/profissional)
- `vitorias`, `derrotas`, `empates`: Contadores de resultados

3.4.2 *Classe Equipe*

A classe `Equipe` implementa o padrão Composite, agrupando múltiplos `Jogadores`.

```
class Equipe {
private:
    string nome;
    vector<Jogador> jogadores;

public:
    Equipe(string nome);

    void adicionarJogador(const Jogador& j);
    void removerJogador(const string& nomeJogador);
    bool temJogador(const string& nomeJogador) const;

    int getTamanho() const;
    vector<Jogador> getJogadores() const;

    // Estat sticas agregadas
    int getTotalVitorias() const;
    double getMediaIdade() const;
};
```

Listing 2: Definição da classe Equipe

3.4.3 *Classe Partida*

Representa um confronto entre dois participantes.

```
class Partida {
private:
    string participante1;
    string participante2;
    ResultadoPartida resultado;
    string data;

public:
    Partida(string p1, string p2);

    void setResultado(ResultadoPartida r);
    ResultadoPartida getResultado() const;

    string getParticipante1() const;
    string getParticipante2() const;
    string getData() const;

    bool foiRealizada() const;
```

```
};
```

Listing 3: Definição da classe Partida

3.4.4 *Classe Torneio*

Responsável pela orquestração completa da competição.

```
class Torneio {
private:
    string nome;
    ModalidadeTorneio modalidade;
    vector<string> participantes;
    vector<Partida> partidas;

public:
    Torneio(string nome, ModalidadeTorneio mod);

    void adicionarParticipante(string nome);
    void removerParticipante(string nome);

    void gerarRodadas();
    vector<Partida> getRodada(int numero) const;

    void registrarResultado(int indexPartida,
                            ResultadoPartida resultado);

    int getNumeroRodas() const;
};
```

Listing 4: Definição da classe Torneio

Modalidades suportadas:

- **PONTOS_CORRIDOS**: Round-robin com todas as equipes jogando entre si
- **MATA_MATA**: Eliminação progressiva em chaves

3.4.5 *Classe Ranking*

Mantém classificação dinâmica dos participantes.

```
class Ranking {
private:
    map<string, int> pontuacoes;
    vector<pair<string, int>> classificacao;

public:
```

```
    void adicionarPontuacao(string participante, int pts);
    void gerar();
    void exibir() const;

    int getPosicao(string participante) const;
    int getPontuacao(string participante) const;

    bool exportar(string nomeArquivo) const;
    bool importar(string nomeArquivo);
};
```

Listing 5: Definição da classe Ranking

Critérios de ordenação:

1. Pontuação total (primário)
2. Número de vitórias (critério de desempate)
3. Saldo de confrontos (segundo desempate)

3.4.6 Classe Persistencia

Abstrai operações de I/O em arquivos CSV.

```
class Persistencia {
public:
    // M todos est ticos para opera  es de I/O
    static bool salvarJogadores(
        const vector<Jogador>& jogadores,
        const string& nomeArquivo);

    static vector<Jogador> carregarJogadores(
        const string& nomeArquivo);

    static bool salvarRanking(
        const Ranking& ranking,
        const string& nomeArquivo);

    static bool salvarPartidas(
        const vector<Partida>& partidas,
        const string& nomeArquivo);
};
```

Listing 6: Definição da classe Persistencia

3.5 Implementação de Conceitos de OO

3.5.1 Encapsulamento

O sistema implementa forte encapsulamento através de:

- **Atributos privados**: Todos os dados são privados
- **Acesso controlado**: Getters e setters para interface pública
- **Validação**: Métodos validam entrada antes de modificar estado
- **Proteção de invariantes**: Garante consistência interna

Exemplo de encapsulamento em Jogador:

```
class Jogador {
private:
    int idade;  // Atributo privado

public:
    // Setter com valida  o
    void setIdade(int novaIdade) {
        if (novaIdade > 0 && novaIdade < 150) {
            idade = novaIdade;
        } else {
            throw invalid_argument("Idade inv lida");
        }
    }

    // Getter para acesso controlado
    int getIdade() const {
        return idade;
    }
};
```

Listing 7: Exemplo de encapsulamento

3.5.2 Herança

Embora o sistema atual não implemente herança direta, a arquitetura foi projetada para permitir extensão futura através de herança. Exemplo de como poderia ser expandida:

```
// Classe base para participantes
class Participante {
protected:
    string nome;
    int pontuacao;
```

```
public:
    virtual int calcularBonus() = 0;
    virtual string getTipo() const = 0;
};

// Specializa o para jogadores
class JogadorComBonus : public Participante {
private:
    TipoJogador tipo;

public:
    int calcularBonus() override {
        return tipo == PROFISSIONAL ? 10 : 0;
    }
};
```

Listing 8: Exemplo de herança potencial

3.5.3 *Polimorfismo*

O polimorfismo dinâmico é implementado através de métodos virtuais (em extensões futuras). O polimorfismo paramétrico é alcançado via templates:

```
// Template gen rico para ordena  o
template <typename T>
class Ordenador {
public:
    static void ordenar(vector<T>& items) {
        sort(items.begin(), items.end());
    }
};

// Uso
vector<int> numeros = {3, 1, 4, 1, 5};
Ordenador<int>::ordenar(numeros);

vector<string> nomes = {"Ana", "Bruno", "Carla"};
Ordenador<string>::ordenar(nomes);
```

Listing 9: Exemplo de polimorfismo paramétrico

3.5.4 *Templates C++*

Templates são utilizados para criar código genérico e reutilizável:

```
// Template para gerenciar cole  o gen rica
```

```cpp
template <typename T>
class Gerenciador {
private:
    vector<T> items;

public:
    void adicionar(const T& item) {
        items.push_back(item);
    }

    T obter(int index) const {
        if (index >= 0 && index < items.size()) {
            return items[index];
        }
        throw out_of_range(" ndice  inv lido");
    }

    int tamanho() const {
        return items.size();
    }

    void listar() const {
        for (const auto& item : items) {
            cout << item << endl;
        }
    }
};
```

Listing 10: Exemplo de template para container genérico

3.5.5 STL (Standard Template Library)

O sistema faz uso extensivo da STL:

- `vector<T>`: Para armazenar coleções dinâmicas (jogadores, partidas, equipes)
- `string`: Para manipulação de strings com facilidade
- `map<K,V>`: Para associar pontuações aos participantes
- `algorithm`: Funções como `sort`, `find`, `transform`
- `iostream`: Para operações de entrada/saída
- `fstream`: Para manipulação de arquivos
- `iomanip`: Para formatação de saída

Exemplo de uso da STL:

```cpp
#include <vector>
#include <algorithm>
#include <map>

// Usando vector para cole  o din mica
vector<Jogador> jogadores;

// Adicionando elementos
jogadores.push_back(Jogador("Ana", 22, "Futebol",
                            PROFISSIONAL));

// Encontrando elemento
auto it = find_if(jogadores.begin(),
                  jogadores.end(),
                  [](const Jogador& j) {
                      return j.getNome() == "Ana";
                  });

// Usando map para associa  es
map<string, int> pontuacoes;
pontuacoes["Ana"] = 15;
pontuacoes["Bruno"] = 12;

// Iterando com range-based for (C++11)
for (const auto& [nome, pts] : pontuacoes) {
    cout << nome << ": " << pts << " pontos" << endl;
}
```

Listing 11: Exemplo de uso da STL

3.5.6 Classes Abstratas (Conceitual)

Embora não implementadas como pure virtual, a arquitetura permite interfaces abstratas:

```cpp
// Classe abstrata (interface)
class ArmazenadorDados {
public:
    virtual ~ArmazenadorDados() = default;
    virtual bool salvar(const string& dados) = 0;
    virtual string carregar() = 0;
};

// Implementa  o concreta para CSV
class ArmazenadorCSV : public ArmazenadorDados {
```

```
public:
    bool salvar(const string& dados) override {
        // Implementa o especfica para CSV
        return true;
    }

    string carregar() override {
        // L gica de carregamento
        return "";
    }
};

// Implementa o concreta para JSON (futura)
class ArmazenadorJSON : public ArmazenadorDados {
public:
    bool salvar(const string& dados) override {
        // Implementa o especfica para JSON
        return true;
    }
};
```

Listing 12: Exemplo conceitual de classe abstrata

3.5.7 Persistência de Dados

O sistema implementa persistência robusta através da classe `Persistencia`, que serializa objetos em formato CSV:

```
// Salvar jogadores em arquivo CSV
vector<Jogador> jogadores;
jogadores.push_back(Jogador("Ana Silva", 22,
                            "Futebol", PROFISSIONAL));
Persistencia::salvarJogadores(jogadores,
                              "jogadores.csv");

// Carregar dados do arquivo
vector<Jogador> carregados =
    Persistencia::carregarJogadores("jogadores.csv");

// Salvar ranking
Ranking ranking;
ranking.adicionarPontuacao("Ana Silva", 15);
ranking.gerar();
ranking.exportar("ranking.csv");
```

Listing 13: Exemplo de persistência

4 CONCEITOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS UTILIZADOS E NÃO UTILIZADOS

4.1 Conceitos Implementados

A Tabela 4 apresenta os conceitos de POO implementados no sistema:

Conceito OO	**Implementação**	**Localização**
Encapsulamento	Atributos privados com acesso via métodos públicos	Todas as classes
Classes	Sete classes principais com responsabilidades distintas	Jogador, Equipe, Partida, etc.
Objetos	Instâncias criadas de cada classe	main.cpp
Atributos	Membros de dados que definem estado do objeto	Definidos em cada classe
Métodos	Funções que definem comportamento do objeto	Implementados em todas classes
Construtores	Inicialização padronizada de objetos	Definidos em todas classes
Destrutores	Limpeza de recursos (implícito com STL)	Não necessário - RAII
Composição	Equipe compõe Jogadores, Torneio compõe Partidas	Equipe, Torneio
Agregação	Ranking agrega Jogadores	Ranking
Associação	Relações entre classes distintas	Torneio-Equipe-Jogador
Templates	Código genérico parametrizado por tipo	Gerenciador<T> (conceitual)
STL	Containers e algoritmos reutilizáveis	vector, map, string, algorithm
Enumeradores	Tipos para representar conjuntos finitos	TipoJogador, ModalidadeTorneio
Métodos const	Métodos que não modificam estado	Getters de todas classes
Referências	Passagem eficiente sem cópia	Parâmetros const& em métodos
Validação	Verificação de invariantes	setIdade(), adicionarJogador()

Tabela 4: Conceitos OO Implementados no Sistema

4.2 Conceitos Não Utilizados e Justificativas

A Tabela 5 apresenta conceitos de POO não utilizados e justificativas técnicas:

Conceito OO	**Justificativa de Não Utilização**	**Impacto**
Herança (polimorfismo dinâmico)	Não havia necessidade de hierarquias de classes; composição foi suficiente	Baixo - Aplicação permanece modular
Classes Abstratas (virtuais puras)	Interfaces bem definidas sem necessidade de polimorfismo dinâmico	Baixo - Contrato implícito funcionou
Sobrecarga de Operadores	Não havia operações matemáticas ou comparações naturais para classes	Negligenciável - Não necessário
Operador de Atribuição Customizado	Cópia padrão da STL foi suficiente para necessidades	Baixo - Valores simples
Ponteiros (uso extensivo)	STL gerencia memória; evita vazamentos e complexidade	Positivo - Aumenta segurança
Alocação Dinâmica Manual	RAII da STL eliminou necessidade; reduz erros	Positivo - Melhor prática moderna
Múltipla Herança	Padrão único (composição) foi suficiente; evita complexidade	Baixo - Design simplificado
Namespaces	Projeto pequeno; nomes bem escolhidos evitaram conflitos	Negligenciável - Projeto educacional
Exceções (uso extensivo)	Tratamento básico; validação em setters foi suficiente	Médio - Poderia melhorar robustez
Genéricos (Generics)	Templates C++ não utilizados em nível avançado	Baixo - Não necessário para escopo
Operador delete Customizado	Garbage collection desnecessário em C++ moderno	Positivo - Segurança via RAII
Properties (.NET style)	Não aplicável em C++ padrão	N/A - Linguagem diferente

Tabela 5: Conceitos OO Não Utilizados e Justificativas

5 DISCUSSÃO E CONCLUSÕES

5.1 Análise da Solução

A solução desenvolvida demonstra a aplicação bem-sucedida dos princípios da Programação Orientada a Objetos para resolver o problema de gestão de torneios esportivos. O trabalho evidencia modularidade nas classes, possibilidade de futuras extensões e reutilização do código por meio da STL. Além disso, garante segurança de tipos, desempenho adequado e uma camada de persistência clara, permitindo fácil evolução do sistema.

5.2 Resultados Obtidos

O sistema implementado alcançou todos os objetivos propostos:

Objetivo	**Resultado**	**Status**
Aplicar OOP	7 classes, encapsulamento, composição	Completo
Arquitetura modular	Separação clara de responsabilidades	Completo
Persistência robusta	CSV com integridade de dados	Completo
Padrões de projeto	Composite, Singleton (conceitual)	Completo
Código limpo	Convenções de nomenclatura, documentação	Completo
Validação	Testes de casos de uso principais	Completo

Tabela 6: Matriz de Alcance de Objetivos

5.3 Limitações Identificadas

Durante o desenvolvimento foram observadas algumas limitações, tais como o uso do formato CSV para armazenamento, que não é ideal para dados mais complexos, a escolha por uma interface de linha de comando que pode restringir a usabilidade, a simulação de partidas baseada apenas em sorte ao invés de habilidades mais realistas e a validação de erros realizada de forma básica. Além disso, o algoritmo de geração de rodadas, apesar de funcional, pode ser otimizado em versões futuras para melhorar a performance e a experiência do usuário.

6 CONSIDERAÇÕES PESSOAIS

Este projeto foi uma oportunidade excepcional de consolidar conhecimentos de Programação Orientada a Objetos através de uma aplicação prática e completa. Durante o de-

senvolvimento, várias lições importantes foram aprendidas:

6.1 Aprendizados Técnicos

Durante o desenvolvimento deste projeto, ficou evidente a importância do design prévio (UML) na qualidade do código final. Investir tempo em modelagem economiza tempo em refatoração posteriormente. Também foi possível perceber que a reutilização via STL é uma melhor prática moderna que reduz bugs e aumenta eficiência, sendo muito mais vantajoso do que implementar soluções próprias. Ademais, encapsulamento não diz respeito apenas ao acesso privado ou público aos atributos, mas sim à comunicação clara entre componentes e à prevenção de efeitos colaterais. Finalmente, a composição demonstrou ser frequentemente mais flexível que a herança, e o design inicial evitou hierarquias complexas, mantendo a simplicidade e facilitando futuras manutenções.

6.2 Aprendizados sobre Engenharia de Software

No que se refere à engenharia de software, ficou claro que princípios SOLID não são conceitos abstratos, mas sim diretrizes com impacto prático imediato na manutenibilidade do código. Percebe-se ainda que a separação de responsabilidades, como a camada de persistência abstraída, permite alterações futuras com menor risco. A documentação clara do contrato de cada classe, por meio de comentários e exemplos, mostra-se tão importante quanto a própria implementação. Por fim, a experiência reforçou que a simplicidade geralmente traz mais benefícios do que o excesso de generalização, devendo-se evitar um design muito sofisticado para situações que ainda não existem no escopo do projeto.

6.3 Reflexão Pessoal

O desenvolvimento deste sistema solidificou a compreensão de que Programação Orientada a Objetos é mais uma filosofia de design do que um simples conjunto de recursos da linguagem. Elementos como classes, encapsulamento e composição funcionam como ferramentas, sendo o verdadeiro valor extraído da forma como são utilizados para criar sistemas elegantes, compreensíveis e fáceis de manter. Além disso, a experiência de trabalhar em um projeto "real"mesmo em um contexto educacional trouxe uma perspectiva valiosa sobre decisões de design, escolhas técnicas e a importância da comunicação clara por meio do código desenvolvido.

7 DIVISÃO DO TRABALHO

Como o trabalho foi realizado individualmente, todas as atividades foram executadas por Thiago dos Santos.

Atividade	**Responsável**
Todas as etapas do projeto (requisitos, modelagem, implementação, testes, documentação, apresentação)	Thiago dos Santos

Tabela 7: Divisão de Trabalho - Projeto Individual

Não se aplicam reuniões de grupo, pois toda a execução foi individual.

8 AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos ao Professor da disciplina de Programação Avançada pela orientação, pelo feedback construtivo e por toda dedicação na preparação do material de qualidade. Agradeço também aos colegas de ensino que contribuíram com o desenvolver das aulas. Registro minha gratidão à comunidade de desenvolvimento C++ por manter elevados padrões de qualidade e compartilhar conhecimento por meio de documentação aberta. Sou grato, ainda, pelo espírito de colaboração demonstrado durante as semanas de desenvolvimento e pela infraestrutura, ambiente e suporte fornecidos pela instituição (UTFPR), que foram fundamentais para a conclusão deste projeto.