Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA E SISTEMAS

Relatório da Meta 1

Relatório de Trabalho Prático em Engenharia Informática

no âmbito da Unidade Curricular de Programação Orientada a Objetos

Autor

Celso André Ferreira Jordão

n.º 2003008910

Pedro Miguel Ribeiro Girão

n.º 2021133806

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, Novembro de 2025

RELATÓRIO META 1 - SIMULADOR DE JARDIM

Programação Orientada a Objetos 2025/2026

Autores: Celso Jordão - 2003008910, Pedro Girão - 2021133806
Data: 2 de Novembro de 2025

ÍNDICE

1 INTRODUÇÃO

Este relatório documenta o desenvolvimento da primeira meta do trabalho prático: um simulador de jardim baseado em turnos implementado em C++. O sistema permite ao utilizador controlar um jardineiro virtual que pode plantar, colher, usar ferramentas e gerir um jardim representado por uma grelha bidimensional.

1.1 Objetivos da Meta 1

Os objetivos principais desta primeira entrega foram:

- **Validação Completa de Comandos**: Implementar *parsing* e validação robusta de todos os comandos especificados no enunciado
- **Estruturação em Classes**: Planear e definir todas as classes necessárias seguindo princípios de orientação a objetos
- **Organização do Código**: Separar código em ficheiros `.h` e `.cpp` com *includes* devidamente organizados
- **Estruturas de Dados**: Implementar a grelha do jardim sem recurso a coleções da biblioteca standard

1.2 Estado Atual

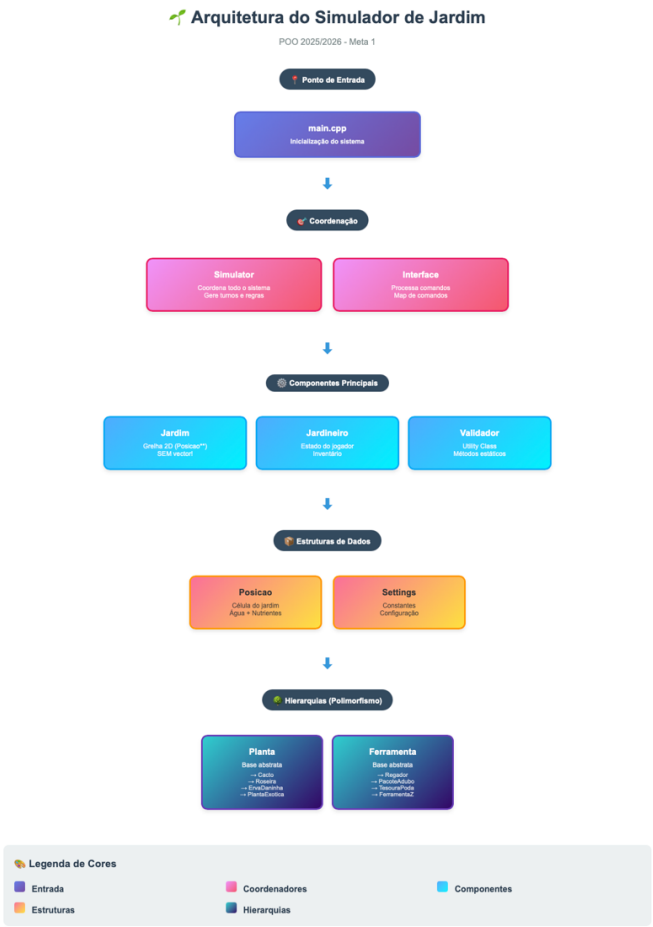
Nesta primeira meta, foi implementado todo o sistema de validação de comandos, a estrutura completa de classes e a alocação dinâmica da grelha do jardim. O comportamento completo das plantas e ferramentas será desenvolvido na Meta 2.

---

2 ARQUITETURA DO SISTEMA

2.1 Visão Geral

O sistema foi estruturado em múltiplas camadas seguindo rigorosamente os princípios de Programação Orientada a Objetos:



2.2 Responsabilidades das Classes

Classe	Responsabilidade
Simulator	Coordena todos os componentes, gere turnos, valida regras de jogo
Interface	Processa e valida comandos do utilizador
Validador	Centraliza todas as validações de entrada
Jardim	Gere grelha 2D, renderização visual
Posicao	Representa célula individual com recursos
Jardineiro	Estado do jogador, inventário
Planta	Base abstrata para vegetação
Ferramenta	Base abstrata para itens

Tabela 1 - Responsabilidades de classes

2.3 Padrões de Design Aplicados

Factory Method: Utilizado em `Simulator` para criar plantas e ferramentas:

```
Planta* Simulator::criaPlanta(char tipo) {
    switch (tipo) {
        case 'c': return new Cacto();
        case 'r': return new Roseira();
        case 'e': return new ErvaDaninha();
        case 'x': return new PlantaExotica();
    }
}
```

Strategy Pattern: Sistema de comandos com `std::map`:

```
std::map<std::string, std::function<bool(istringstream&)>> comandos;
comandos["jardim"] = [this](auto& iss) { return cmdJardim(iss); };
```

Template Method: Classes base `Planta` e `Ferramenta` definem esqueleto, subclasses implementam detalhes.

Utility Class: `Validador` fornece métodos estáticos reutilizáveis sem estado.

---

3 DECISÕES CRÍTICAS DE IMPLEMENTAÇÃO

3.1 Grelha do Jardim - Array 2D Dinâmico

Decisão: Implementar usando `Posicao**` (ponteiro para array de ponteiros).

Justificação:

1. O enunciado **proíbe explicitamente** o uso de `std::vector` para armazenamento do solo
2. Permite dimensões variáveis definidas em runtime (1x1 até 26x26)
3. Aloca apenas a memória estritamente necessária
4. Acesso eficiente em O(1): `grelha[linha][coluna]`

Implementação:

```
class Jardim {
    Posicao** grelha;  // Array 2D dinâmico
    int linhas, colunas;

    Jardim(int l, int c) {
        grelha = new Posicao*[linhas];
        for (int i = 0; i < linhas; i++) {
            grelha[i] = new Posicao[colunas];
```

```
        }
    }

    ~Jardim() {
        for (int i = 0; i < linhas; i++) {
            delete[] grelha[i];
        }
        delete[] grelha;
    }
};
```

Alternativas Consideradas:

- Array unidimensional com cálculo de índice: rejeitado por complexidade desnecessária
- Alocação estática 26x26: rejeitado por desperdício de memória

3.2 Sistema de Comandos com Map

Decisão: Usar `std::map<std::string, std::function<>>` para despachar comandos.

Antes (abordagem ingénua):

```
if (cmd == "jardim") return cmdJardim(iss);
else if (cmd == "avanca") return cmdAvanca(iss);
else if (cmd == "lplantas") return cmdLPlantas(iss);
// ... 20+ comandos em cascata
```

Problemas: Código extenso, busca O(n) linear, difícil manutenção.

Depois (com map):

```
std::map<string, function<bool(istringstream&)>> comandos;

void inicializaComandos() {
    comandos["jardim"] = [this](auto& iss) { return cmdJardim(iss); };
    comandos["avanca"] = [this](auto& iss) { return cmdAvanca(iss); };
    // ...
}

bool processaComando(const string& linha) {
    auto it = comandos.find(cmd);
    if (it != comandos.end()) {
        return it->second(iss);  // Executa função
    }
}
```

3.3 Classe Validador (Utility Class)

Decisão: Centralizar todas as validações numa classe separada com métodos estáticos.

Motivação: Separação de responsabilidades - `Interface` processa comandos, `Validador` valida dados.

Implementação:

```
class Validador {
public:
    static bool validaCoordenada(const string& coord, int& l, int& c);
    static bool validaInt(const string& str, int& valor);
    static bool validaTipoPlanta(char tipo);
    static bool validaDimensoesJardim(int l, int c);
    // ...
private:
    Validador() = delete;  // Não pode instanciar
};
```

3.4 Hierarquias com Classes Abstratas

Decisão: Usar classes base abstratas com métodos virtuais puros.

```
class Planta {
protected:
    int aguaAcumulada, nutrientesAcumulados;
    Beleza beleza;

public:
    virtual void avancaInstante(Posicao&, Jardim&, int, int) = 0;
    virtual bool deveMorrer() const = 0;
    virtual char getSimbolo() const = 0;
    virtual ~Planta() {}  // Destrutor virtual!
};
```

---

4 ORGANIZAÇÃO E BOAS PRÁTICAS

4.1 Estrutura de Ficheiros

```
SimuladorJardim/
├── core/
│   ├── Simulator.h / .cpp
│   ├── Interface.h / .cpp
│   └── Validador.h / .cpp
├── jardim/
│   ├── Jardim.h / .cpp
│   └── Posicao.h / .cpp
├── jardineiro/
│   └── Jardineiro.h / .cpp
```

```
├── plantas/
│   ├── Planta.h / .cpp
│   └── [4 subclasses]
├── ferramentas/
│   ├── Ferramenta.h / .cpp
│   └── [4 subclasses]
├── config/
│   └── Settings.h
├── main.cpp
└── CMakeLists.txt
```

Vantagens: Modularidade, fácil navegação, separação por funcionalidade.

4.2 Organização de Includes

Princípio aplicado: Header próprio → Headers projeto → Headers STL → Headers sistema

Exemplo (Jardim.cpp):

```
#include "Jardim.h"           // 1° - Próprio

// Headers do projeto
#include "Posicao.h"
#include "Settings.h"

// Headers STL
#include <iostream>
#include <cstdlib>
```

Benefícios: Consistência, fácil leitura, evita dependências circulares.

4.3 Forward Declarations

Uso estratégico para minimizar dependências:

```
// Jardim.h
class Planta;      // Forward declaration
class Ferramenta;  // Forward declaration

class Jardim {
    // Apenas ponteiros - não precisa definição completa
};
```

Vantagem: Compilação mais rápida, menos recompilações.

4.3.1 4.4 Gestão de Memória

Princípio RAII aplicado consistentemente:

```
class Jardim {
    Posicao** grelha;
```

```
public:
    Jardim(int l, int c) {
        grelha = new Posicao*[l];  // Alocação
        // ...
    }

    ~Jardim() {
        // Libertação automática
        for (int i = 0; i < linhas; i++) {
            delete[] grelha[i];
        }
        delete[] grelha;
    }

    // Delete copy constructor/assignment
    Jardim(const Jardim&) = delete;
    Jardim& operator=(const Jardim&) = delete;
};
```

Garantias: Sem memory leaks, sem double-free, ownership claro.

---

5 VALIDAÇÃO DE COMANDOS

5.1 Comandos Implementados

Total: 20 comandos validados completamente

Categoria	Comandos	Validações
Criação	`jardim <l> <c>`	Dimensões 1-26, inteiros positivos
Tempo	`avanca [n]`	Inteiro positivo opcional
Listagem	`lplantas`, `lplanta <pos>`, `larea`, `lsolo <pos> [raio]`, `lferr`	Coordenadas, raio não-negativo
Ações	`colhe <pos>`, `planta <pos> <tipo>`, `larga`, `pega <num>`, `compra <tipo>`	Coordenadas, tipos válidos, números
Movimento	`e`, `d`, `c`, `b`, `entra <pos>`, `sai`	Coordenadas
Persistência	`grava <nome>`, `recupera <nome>`, `apaga <nome>`	Nomes válidos

Categoria	Comandos	Validações
Ficheiros	`executa <ficheiro>`	Nome ficheiro válido
Sistema	`fim`	-

5.2 Exemplos de Validação

Coordenadas (formato "aa" a "zz"):

```
bool validaCoordenada(const string& coord, int& l, int& c) {
    if (coord.length() != 2) return false;

    char linha = tolower(coord[0]);
    char coluna = tolower(coord[1]);

    if (linha < 'a' || linha > 'z') return false;
    if (coluna < 'a' || coluna > 'z') return false;

    l = linha - 'a';  // Conversão 'a'=0, 'b'=1, ...
    c = coluna - 'a';
    return true;
}
```

Tipos de Planta/Ferramenta:

```
bool validaTipoPlanta(char tipo) {
    tipo = tolower(tipo);
    return tipo == 'c' || tipo == 'r' ||
           tipo == 'e' || tipo == 'x';
}
```

5.3 Mensagens de Erro

Todas as validações fornecem feedback claro:

```
> jardim -5 100
[ERRO] Dimensoes invalidas (min: 1x1, max: 26x26)

> planta zz q
[ERRO] Tipo invalido. Use: c (cacto), r (roseira), e (erva), x
(exotica)

> colhe aa
[OK] Comando valido (nao implementado)
```

7. Conclusão

A Meta 1 estabeleceu uma fundação sólida para o projeto. Os principais objetivos foram cumpridos:

Arquitetura Robusta: Sistema bem estruturado em classes com responsabilidades claras, seguindo princípios SOLID.

Validação Completa: Todos os 20 comandos validados com feedback apropriado ao utilizador.

Decisões Técnicas Sólidas: Grelha dinâmica sem `vector`, sistema de comandos eficiente com `map`, validações centralizadas.

Código Organizado: Separação clara em ficheiros, includes consistentes, hierarquias bem definidas.

Preparação para Meta 2: Estrutura permite adicionar comportamentos sem reestruturação, interfaces prontas, TODOs claramente marcados.

A implementação demonstra domínio dos conceitos de orientação a objetos, gestão adequada de memória dinâmica e capacidade de organizar código complexo de forma manutenível e extensível.