



Instituto Superior de Engenharia

Politécnico de Coimbra

SIMULADOR DE UM JARDIM

Programação Orientada a Objetos

Trabalho Prático

2025/2026

Lucas Golobovante – 2023140728

Diogo Ferreira - 2023141377

Contents

1. Introdução	2
2. Arquitetura do Sistema	2
2.1 Organização dos Ficheiros	2
2.2 Estruturas de Dados e Gestão de Memória	2
2.3 Padrão de Desenho: Command.....	3
2.4 Configuração Centralizada (Settings).....	3
3. Modelação das Entidades e Polimorfismo	3
3.1 Hierarquia de Plantas (Polimorfismo)	3
3.2 O Jardineiro (Encapsulamento)	4
4. Gestão de Itens e Padrões de Desenho	4
4.1 Funcionalidade Específica das Ferramentas.....	4
4.2 Identificação e Durabilidade das Ferramentas	5
4.3 Padrão de Desenho	5
5. Algoritmos e Fluxo de Execução	5
5.1 Avanço do Tempo (avanca).....	7
5.2 Algoritmo de Visualização (larea).....	7
5.3 Movimentação e Coleta Automática (move).....	7
5.4 Persistência de Dados (grava e recupera)	7
6. Conclusão e Análise Crítica.....	8
6.1 Objetivos Atingidos	8
6.2 Dificuldades e Soluções	8

1. Introdução

O presente projeto consiste no desenvolvimento de um simulador de um Jardim em linguagem C++, aplicando os conceitos da Programação Orientada a Objetos (POO). O objetivo principal é simular um ecossistema biológico simplificado onde um Jardineiro interage com diversas espécies de plantas e utiliza ferramentas para a sua manutenção.

O sistema foi desenhado para respeitar restrições, nomeadamente:

1. Gestão Manual de Memória: Proibição do uso de contentores da biblioteca standard (como `std::vector`) para a representação da grelha do jardim.
2. Polimorfismo: Implementação de comportamentos distintos para entidades biológicas e ferramentas.
3. Persistência: Capacidade de gravar e recuperar o estado completo da simulação.

2. Arquitetura do Sistema

O projeto encontra-se estruturado de forma modular, separando a lógica de controlo, a representação de dados e a interação com o utilizador.

2.1 Organização dos Ficheiros

O código foi dividido em pasta:

- Core: Contém o motor do sistema. Inclui a classe Jardim (gestão da grelha), Simulador (controlador principal e game loop) e Settings (configurações globais).
- Entidades: Define os agentes vivos (Jardineiro e a hierarquia de Planta).
- Items: Contém a hierarquia de objetos inanimados (Ferramenta e subclasses).
- Comandos: Implementa o padrão de desenho Command, encapsulando cada ação do utilizador numa classe própria.

2.2 Estruturas de Dados e Gestão de Memória

Para cumprir o requisito de não utilizar contentores da biblioteca standard na grelha:

- Matriz Dinâmica: A área do jardim é representada por uma matriz dinâmica de ponteiros (`Posicao** grelha`). A classe Jardim é responsável pela alocação desta memória no construtor e pela sua correta libertação no destrutor, prevenindo memory leaks.
- Célula (`Posicao`): Cada posição da grelha é um objeto que agrega ponteiros para Planta e Ferramenta, com o uso de ponteiros pode-se verificar facilmente se uma posição está vazia (`nullptr`) ou ocupada, além de facilitar a troca dinâmica de objetos durante a execução.

2.3 Padrão de Desenho: Command

A interação com o utilizador segue o padrão Command.

- Abstração: Existe uma classe base abstrata Comando com um método virtual puro `executa()`.
- Fábrica de Comandos: O método `Simulador::criaComando` atua como uma fábrica, recebendo uma string de input, analisando-a e instanciando a subclasse apropriada (ex: `new Avanca`, `new Colhe`).
- Esta abordagem permite adicionar novos comandos sem alterar a lógica do ciclo principal do simulador.

2.4 Configuração Centralizada (Settings)

Para evitar o uso de "números mágicos" no código, foi criada a classe Settings. Esta classe contém apenas constantes estáticas públicas. Isto permite ajustar o equilíbrio do jogo num único ficheiro, sem necessidade de recompilar múltiplas classes.

3. Modelação das Entidades e Polimorfismo

O núcleo da simulação está na hierarquia de classes que representa os seres vivos e na gestão do utilizador.

3.1 Hierarquia de Plantas (Polimorfismo)

Todas as plantas herdam da classe base abstrata Planta. Esta classe define os atributos comuns (`aguaInterna`, `nutrientesInternos`, `beleza`, `instantesDeVida`) e impõe um contrato através de métodos virtuais puros, nomeadamente `agir()` e `getChar()`.

O motor de simulação (Simulador) trata todas as plantas de forma genérica, invocando o método `agir()` sobre ponteiros do tipo `Planta*`. No entanto, graças ao Polimorfismo, o comportamento executado depende da espécie concreta:

- Roseira: Apresenta uma lógica complexa de sobrevivência, morrendo se tiver excesso ou falta de nutrientes. Reproduz-se para uma posição adjacente aleatória quando atinge um patamar de nutrientes.
- Cacto: Implementa uma mecânica de absorção de água baseada numa percentagem do solo, adaptada a ambientes áridos, e morre se o solo estiver demasiado húmido (excesso de água).
- Erva Daninha: Tem um ciclo de vida finito, morrendo por velhice após um número fixo de instantes. Reproduz-se agressivamente, espalhando-se rapidamente pelo jardim.
- Planta Exótica (Bambu Dourado): Inspirada na rápida propagação natural desta espécie, a implementação define um comportamento de expansão em cruz (cima, baixo, esquerda, direita). Ao acumular recursos hídricos suficientes, a planta clona-se para os vizinhos disponíveis, simulando o alastramento invasivo de uma floresta de bambu.

3.2 O Jardineiro (Encapsulamento)

A classe Jardineiro encapsula toda a interação do jogador com o mundo.

- **Gestão de Inventário:** O jardineiro possui um vetor de ponteiros (`std::vector<Ferramenta*>`) para representar a mochila e um ponteiro dedicado (`ferramentaNaMao`) para o item ativo.
- **Controlo de Ações:** Para cumprir as regras do enunciado, o jardineiro mantém contadores internos (`movimentosTurno`, `colheitasTurno`, etc.) que são decrementados a cada ação e reiniciados no início de cada novo instante. Isto impede que o jogador realize ações infinitas num único turno.

4. Gestão de Itens e Padrões de Desenho

4.1 Funcionalidade Específica das Ferramentas

Cada ferramenta concreta herda da classe base `Ferramenta` e implementa a sua própria versão do método virtual `usar(Jardim&, int, int)`. O comportamento varia consoante o tipo de objeto:

- **Regador (g):** Atua sobre a hidratação do solo. Ao ser utilizado, transfere uma quantidade fixa de água (definida em `Settings`) para a `Posicao` onde o jardineiro se encontra. Possui um atributo interno de capacidade que diminui a cada uso. Quando esta capacidade se esgota (verificada pelo método `deveQuebrar`), o regador é considerado inutilizável e removido do sistema.
- **Pacote de Adubo (a):** Funciona de forma parecida ao Regador, mas atua sobre os nutrientes do solo. Aumenta o nível de fertilização da posição atual e consome a sua própria reserva interna até se esgotar e desaparecer.
- **Tesoura de Poda (t):** Implementa uma lógica de seleção. Ao ser usada, verifica se existe uma planta na posição atual. Se existir, consulta o atributo de beleza da planta (`planta->getBeleza()`). A tesoura apenas executa o corte (removendo a planta da memória) se a beleza for classificada como feia (ex: Erva Daninha). Caso contrário, a planta é poupada. Sendo uma ferramenta infinita, o método `deveQuebrar` retorna sempre `false`.
- **FerramentaZ / Drone de Rega (z):** Esta ferramenta especial distingue-se por ter um efeito de área. O seu método `usar` implementa dois ciclos `for` (de -1 a 1) para percorrer as coordenadas relativas ao redor do jardineiro. Isto permite irrigar simultaneamente a posição do utilizador e as 8 posições vizinhas (caso sejam válidas dentro dos limites do jardim), demonstrando a capacidade do sistema em manipular múltiplas células numa única ação.

4.2 Identificação e Durabilidade das Ferramentas

A classe base Ferramenta implementa dois mecanismos fundamentais:

1. Identificação Única: Foi utilizado um atributo estático (static int proxNumSerie) partilhado por todas as instâncias. No construtor de cada ferramenta, este contador é atribuído ao ID da instância e incrementado, garantindo que cada ferramenta gerada (seja comprada ou encontrada) possui um número de série único e irrepetível.
2. Desgaste: O método virtual deveQuebrar() permite distinguir ferramentas finitas de infinitas. O Regador e o Adubo retornam true quando a sua capacidade se esgota, enquanto a Tesoura utiliza a implementação base que retorna sempre false.

4.3 Padrão de Desenho

Para resolver a necessidade de duplicar objetos polimórficos sem conhecer o seu tipo concreto (necessário na reprodução das plantas e na cópia de segurança do jardim), foi aplicado o Padrão Prototype.

- Cada classe concreta implementa o método clone() const.
- Este método invoca o construtor de cópia da própria classe (ex: return new Roseira(*this)).
- Assim, quando o Jardim precisa de ser copiado para um Save, ele percorre a grelha e chama p->clone(), obtendo uma cópia exata do objeto (seja ele um Cacto ou uma Roseira) sem precisar de fazer casts ou verificações de tipo.

5. Algoritmos e Fluxo de Execução

O motor do simulador baseia-se num ciclo infinito (Game Loop) implementado no método Simulador::run(), que aguarda input do utilizador, processa-o e atualiza o estado do mundo.

Aqui tem uma tabela com todos os comandos do projeto:

Categoria	Comando	Classe	Descrição Técnica
Sistema	jardim	JardimCmd	Aloca a matriz dinâmica da grelha.
	avanca	Avanca	Desencadeia o ciclo biológico (agir) e reseta turnos.
	executa	Executa	Lê ficheiro e injeta linhas no processador de comandos.
	fim	-	Termina o game loop e invoca o destrutor do Simulador.

Categoria	Comando	Classe	Descrição Técnica
Persistência	grava	Grava	Cria uma Deep Copy do jardim e guarda-a em memória.
	recupera	Recupera	Restaura um estado anterior (apaga o atual).
	apaga	Apaga	Remove um save da memória (delete).
Ações	planta	PlantaCmd	Instancia uma nova planta na posição indicada.
	colhe	Colhe	Remove planta e liberta memória (respeita limites).
	compra	Compra	Adiciona ferramenta ao vetor do Jardineiro.
	pega/larga	Pega/Larga	Gere a troca entre inventário e mão.
Movimento	move	Move	Altera coordenadas e apanha itens automaticamente.
	entra/sai	Entra/Sai	Gere a flag noJardim do Jardineiro.
Informação	larea	LArea	Varrimento da matriz com prioridade de visualização.
	Isolo	LSolo	Lista detalhes num raio quadrado em torno de um ponto.
	lplantas	LPlantas	Itera sobre a grelha listando apenas seres vivos.
	lferr	LFerr	Lista o conteúdo do vetor de inventário.

5.1 Avanço do Tempo (avanca)

A passagem do tempo não é apenas um incremento numérico. O comando Avanca desencadeia uma atualização em cadeia de todo o ecossistema:

1. Reset de Ações: Os contadores de turno do Jardineiro (movimentos, plantações, colheitas) são reiniciados, permitindo novas ações no novo instante.
2. Se o jardineiro tiver uma ferramenta equipada (ex: Regador), esta é ativada automaticamente na posição atual.
3. Ciclo Biológico: É invocado o método `jardim->atualizarPlantas()`, que percorre a grelha e solicita a cada planta viva que execute o seu método `agir()`. É neste momento que ocorrem os fenómenos de crescimento, alimentação, morte e reprodução.

5.2 Algoritmo de Visualização (larea)

Para gerar o relatório da área visível, o algoritmo percorre a matriz de posições. Em cada célula, verifica a existência de entidades seguindo uma hierarquia de informação:

1. Verifica se existe uma Planta e concatena a sua descrição (espécie e beleza).
2. Verifica se existe uma Ferramenta no chão.
3. Verifica se o Jardineiro está nessa posição específica.

Esta abordagem garante que o utilizador tem uma visão completa do conteúdo de cada célula, mesmo que visualmente (no comando desenha) apenas o carácter mais prioritário seja mostrado.

5.3 Movimentação e Coleta Automática (move)

O comando de movimentação implementa uma lógica de interação automática. Após validar as coordenadas e atualizar a posição do jardineiro (`posLinha`, `posColuna`), o sistema verifica imediatamente o solo da nova posição.

Se existir uma ferramenta no chão (`pos->getFerramenta() != nullptr`), esta é removida do solo e transferida para o inventário do jardineiro através do método `apanhaFerramenta()`. Isto cumpre o requisito de coleta automática sem necessidade de um comando explícito.

5.4 Persistência de Dados (grava e recupera)

A funcionalidade de gravar o jogo não se limita a guardar dados simples. Envolve a criação de uma cópia profunda de todo o estado do jardim para evitar que alterações no jogo atual afetem os jogos salvos.

- Gravar: O comando Grava invoca o construtor de cópia do Jardim. Este construtor aloca uma nova matriz `Posicao**` e percorre o jardim original, clonando cada Planta e Ferramenta encontrada (usando o padrão Prototype). O novo ponteiro `Jardim*` é guardado num mapa `std::map<string, Jardim*>` saves.
- Recuperar: O comando Recupera localiza o save no mapa, apaga o jardim atual da memória (para evitar leaks) e cria uma nova cópia do jardim guardado para ser o jardim ativo.

6. Conclusão e Análise Crítica

O desenvolvimento deste simulador de jardim permitiu consolidar de forma prática os pilares da Programação Orientada a Objetos em C++.

6.1 Objetivos Atingidos

O projeto cumpre os requisitos propostos no enunciado:

- **Gestão de Memória:** A estrutura de dados baseada em matrizes dinâmicas (`Posicao**`) substituiu eficazmente os contentores STL, com uma gestão rigorosa de alocação e libertação de recursos no destrutor da classe Jardim e Simulador (incluindo a limpeza dos jogos gravados).
- **Mecânicas de Jogo:** Foram implementadas as lógicas de sobrevivência das plantas, a durabilidade das ferramentas e a persistência de dados (grava/recupera).
- **Extensibilidade:** A arquitetura baseada no padrão Command e no uso de interfaces abstratas (Planta, Ferramenta) revelou-se robusta. Adicionar uma nova espécie de planta ou ferramenta exigiria apenas a criação de uma nova classe e a sua inclusão na fábrica de comandos, sem necessidade de alterar o núcleo do simulador.

6.2 Dificuldades e Soluções

A principal dificuldade encontrada residiu na gestão de ponteiros e na garantia de que não existiam fugas de memória (memory leaks) ao substituir objetos na grelha (ex: quando uma planta morre ou é colhida). A solução passou pela implementação cuidadosa dos destrutores e pela verificação constante de ponteiros nulos (`nullptr`) antes de qualquer acesso.

Adicionalmente, a implementação do construtor de cópia para o sistema de saves exigiu o uso do padrão Prototype (`clone()`), uma solução que evitou a complexidade de copiar manualmente objetos de tipos diferentes.