UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Laboratório 5

Simulador de Robôs

MC322 - Programação Orientada a Objetos

Professora: Esther Colombini
PEDs: Angelica Cunha dos Santo / Athyrson Machado Ribeiro / Wladimir Arturo Garces Carrillo

---

1 Descrição Geral

Neste Laboratório 5, você continuará o desenvolvimento do simulador de robôs com foco especial em **organização modular do código**, reutilização de componentes e execução de **missões automatizadas**. Os robôs agora devem ser compostos por subsistemas internos e capazes de executar comportamentos específicos de forma autônoma, sem necessidade de interação direta com o usuário.

1.1 Tópicos Didáticos Abordados

Tema	Incluir?	Forma de abordagem
Abstração	±	Classe abstrata base para robôs (ex: `AgenteInteligente`)
Associação / Agregação	±	Subsistemas como sensores e movimento adicionados ao robô
Composição	±	Robô é composto por módulos de controle e comunicação
Modularização	±	Separar em pacotes: `robo`, `ambiente`, `missao`, etc.
Missões automatizadas	±	Interface `Missao` com classes concretas como `MissaoExplorar`
Arquivos (Obrigatório)	✓	Gravar log de execução da missão

2 Objetivos

Os objetivos principais do Laboratório 5 são os seguintes:

- Praticar abstração e herança com classes abstratas e interfaces.
- Aplicar os conceitos de associação, agregação e composição entre classes.
- Organizar o projeto em pacotes lógicos e coesos.
- Criar missões autônomas controladas por interfaces e subsistemas.
- Registrar logs de execução em arquivos externos.

3 Atividades Propostas

3.1 Reestruturação do Projeto

- Organize seu projeto em pacotes claros, por exemplo:

```
/src/
|-- robo/
|-- ambiente/
|-- comunicacao/
|-- sensores/
|-- missao/
\__ main/

```

- Crie uma classe abstrata `AgenteInteligente` para robôs com comportamento autônomo.

3.2 Classe Abstrata `AgenteInteligente`

A classe `AgenteInteligente` representa um tipo especial de robô com comportamento autônomo, ou seja, que possui uma missão associada e sabe que deve executá-la, mas ainda não define *como* ela será executada. Por esse motivo, trata-se de uma classe **abstrata**, que deve ser estendida por robôs concretos, como `RoboExplorador`, `RoboPatrulheiro`, etc.

- **Herança:** a classe `AgenteInteligente` deve estender a classe `Robo`, herdando seus atributos e comportamentos básicos.
- **Responsabilidade principal:** armazenar uma missão do tipo `Missao` e fornecer um método abstrato `executarMissao()` que será implementado pelas subclasses.
- **Motivação didática:** nem todo robô precisa ser inteligente/autônomo; separar essa especialização ajuda a organizar melhor a hierarquia de classes e o comportamento esperado.

Exemplo de implementação:

```
public abstract class AgenteInteligente extends Robo {
    protected Missao missao;

    public void definirMissao(Missao m) {
        this.missao = m;
    }

    public boolean temMissao() {
        return missao != null;
    }

    public abstract void executarMissao(Ambiente a);
}
```

Listing 1: Classe abstrata AgenteInteligente

Exemplo de robô concreto que herda de `AgenteInteligente`:

```
public class RoboExplorador extends AgenteInteligente {

    @Override
    public void executarMissao(Ambiente a) {
        if (temMissao()) {
            System.out.println("Executando missão exploratória...");
            missao.executar(this, a);
        }
    }
}
```

Listing 2: RoboExplorador que executa a missão

Nota: Esse padrão permite que diferentes tipos de missões (explorar, buscar ponto, patrulhar, etc.) sejam atribuídas dinamicamente a robôs que saibam lidar com elas promovendo flexibilidade, reutilização de código e aplicação de polimorfismo.

3.3 Interface `Missao`

A interface `Missao` define um contrato genérico para tarefas autônomas que um robô pode executar no ambiente. Sua principal função é encapsular a lógica de uma missão como explorar, patrulhar ou buscar um ponto permitindo que robôs autônomos recebam diferentes comportamentos sem precisar conhecer suas implementações concretas.

Definição da interface:

```
public interface Missao {
    void executar(Robo r, Ambiente a);
}
```

Listing 3: Interface Missao

Como funciona:

- A interface é usada por instâncias de `AgenteInteligente`, que armazenam uma referência para uma `Missao`.

- Cada implementação de `Missao` define sua própria lógica de execução, que será chamada por um robô.

Exemplo de missão concreta:

```
public class MissaoExplorar implements Missao {
    public void executar(Robo r, Ambiente a) {
        // Exemplo de movimentação aleatória ou varredura de área
        System.out.println("Robô " + r.getId() + " está explorando...");
        // lógica fictícia de movimentação (pode usar métodos do ambiente)

    }
}
```

Listing 4: MissaoExplorar

Outras sugestões de missões concretas:

- `MissaoPatrulhar` segue um caminho predefinido.
- `MissaoBuscarPonto` move-se até uma coordenada específica.
- `MissaoMonitorar` permanece atento a obstáculos em uma zona.

Observação: O uso dessa interface permite aplicar o princípio de polimorfismo. Cada robô pode executar uma missão diferente usando a mesma chamada: `missao.executar(this, ambiente)`.

3.4 Composição Interna dos Robôs

- Cada robô deve conter subsistemas internos:
  - `ControleMovimento`
  - `GerenciadorSensores`
  - `ModuloComunicacao`
- Os módulos devem ser passados por agregação e permitir composição clara do comportamento do robô.

3.5 Execução de Missões Autônomas

- No menu interativo, permita:
  - Atribuir uma missão a um robô
  - Executar a missão passo a passo
- A execução deve envolver:
  - Movimentação
  - Verificação de sensores
  - Registro de mensagens e estados

3.6 Registro de Logs

- Salve o log da missão em um arquivo de texto com:
  - Posições visitadas
  - Sensores ativados
  - Obstáculos detectados

3.7 Entrada de Dados via Arquivo (Opcional)

Para facilitar a inicialização do simulador e reforçar o uso de arquivos em Java, você pode implementar a leitura de um arquivo de configuração para montar o ambiente, instanciar os robôs e atribuir missões automaticamente.

Exemplo de arquivo de entrada (`config.txt`):

```
AMBIENTE 10 10 3
ROBO Sensoravel R1 1 1 0
ROBO Comunicavel R2 2 2 0
OBSTACULO Arvore 5 5 0
MISSAO R1 EXPLORAR
MISSAO R2 BUSCAR 7 7 0
```

Listing 5: Exemplo de arquivo de configuração

Formato:
Um exemplo do formato é fornecido como segue:

- `AMBIENTE X Y Z` → Define as dimensões do ambiente.
- `ROBO <Tipo> <ID> X Y Z` → Cria um robô de tipo específico com identificação e posição.
- `OBSTACULO <Tipo> X Y Z` → Adiciona um obstáculo na coordenada indicada.
- `MISSAO <ID_robo> <TIPO_MISSAO> [parametros]` → Atribui uma missão a um robô.

Implementação sugerida:

- Crie uma classe utilitária, como `LeitorConfiguracao`, responsável por ler o arquivo e inicializar o sistema.
- Use estruturas de decisão (`switch-case` ou `if`) para interpretar as linhas.
- Reforce o tratamento de exceções para lidar com erros de leitura ou formatos incorretos.

Benefícios:

- Separa os dados da lógica.
- Facilita testes com diferentes cenários.
- Permite gerar novos mapas rapidamente.

Exemplo de uso no `main()`:

```
public static void main(String[] args) {
    Ambiente ambiente = new Ambiente();
    LeitorConfiguracao.carregar("config.txt", ambiente);
    // iniciar menu ou execuções aqui
}
```

4 Avaliação

Critérios de avaliação:

- Criatividade na implementação das classes e suas funcionalidades.
- Ausência de erros de execução e implementação correta das funcionalidades. Interação funcional entre robôs e ambiente.
- Uso da main para efetuar testes das funcionalidades. Output do projeto, o que inclu formatacao da sada de dados e menu de leitura de dados;
- Readme dentro das normas dos laboratórios anteriores.
- Diagrama de classes corretamente desenhado.
- Boas praticas (encapsulamento, modularidade, gets e sets, comentarios no codigo).
- (Bônus) Suporte à leitura do estado inicial a partir de um arquivo de texto formatado, com parsing correto das instruções e inicialização autônoma do simulador.

Observação: A entrada de dados via arquivo é opcional, mas sua implementação correta pode contribuir positivamente na avaliação final (bônus).

5 Entrega

Para a entrega do trabalho considere:

1. **A entrega é realizada exclusivamente via Github** (crie um link da release e submeta (como link) no Google Classroom).
2. Gere um release no Github com a tag no formato *"lab04-RA1-RA2"*.
3. **Prazo de Entrega:** 15/06/25 - 23h59.