O Problema
A partir do código de src2 do repo https://github.com/SW-Engineering-Courses-Karina-Kohl/TCP-Lab-4, observe a implementação de close() em Departamento e Funcionario (implementação da interface AutoCloseable).

Mesmo após chamar deptTI.close() (ou deptTI = null), os funcionários **ainda existem** e podem ser usados (como func1, func2). Isso ocorre porque:

- **As variáveis func1 e func2 no main() ainda mantêm referências** a esses objetos.
- Dentro de Departamento.close(), o trecho funcionario = null; **não faz efeito fora do loop**, porque funcionario é apenas uma **cópia da referência** — não apaga a referência real da lista.

O que fazer para simular uma composição forte?
Se você quer garantir que os funcionários sejam "destruídos" quando o departamento for destruído, você precisa:

1. Encapsular os funcionários — impedir referências externas
- Não permitir que objetos Funcionario sejam criados ou mantidos fora do Departamento.
- Em vez disso, faça o **Departamento** criar e gerenciar os funcionários internamente.
- A ideia é que o Departamento **continue dono dos funcionários**, mas forneça acesso **controlado** para outros componentes (como Projeto).

Passo 1 — Departamento cria os funcionários
- Você não vai mais criar Funcionario diretamente no main(). O Departamento vai ter um método de contratarFuncionario que vai **retornar um objeto Funcionario**, o que é necessário para agregação funcionar, mas você documenta que ele é "owned by Departamento".

Passo 2 — Projeto recebe referência, mas não assume posse
- No Projeto, não muda nada. Ele guarda a referência, mas não gerencia o ciclo de vida.
- projeto.adicionarMembro(funcionario); // funcionário ainda "pertence" ao departamento

Passo 3 — Remover os funcionários dos projetos ao "morrer"
- Quando um Departamento for destruído, os funcionários **devem ser removidos também de todos os projetos onde estavam**.
-
- Formas complexas:
  - Fazer com que cada Funcionario saiba em que projetos ele participa (**complexifica o modelo**), ou
  - Centralizar essa lógica em um **coordenador**, como um SistemaDeRH, que rastreia todos os vínculos.
- Ou mais simples:
  - **Projeto verifica se membros ainda existem**
  - Você pode marcar funcionários como "ativos" ou não (no close() do funcionário, seta ativo = false)

Então

- **Java não permite "destruir" objetos diretamente, mas apenas simular a composição forte .** Você pode ter **composição com Departamento** e **agregação com Projeto**, se:
- O Departamento **controla o ciclo de vida** dos Funcionarios. Encapsule os objetos compostos (não deixe serem criados fora).
- O Projeto **usa referências** a esses objetos, sem controlá-los.
- Gerencie a vida útil internamente. Quando o Departamento é fechado, os Funcionario são desligados e os Projetos passam a vê-los como **inativos** de forma lógica, não física.
    - O que isso significa? Você vai precisar "testar" se um funcionário está ativo ou inativo "manualmente". Não confie no System.gc() para testes — use a lógica de limpeza.

Por que Java não permite composição forte automática

1. Java usa Garbage Collection

Java **não tem controle explícito de destruição de objetos** como em linguagens como C++ (com destructors). Em vez disso, ele usa um **coletor de lixo (Garbage Collector)**, que:

- Libera objetos **somente quando não há mais nenhuma referência viva a eles**;
- Opera de forma **assíncrona e não determinística**;
- **Não permite** que um objeto "delete" outro diretamente da memória.

Composição é um conceito de design, não de memória

A **composição em Java é semântica** — ou seja, ela é expressa por *quem cria, gerencia e usa os objetos*, mas **não há mecanismo automático que "destrua" um objeto contido dentro de outro**.

Java não possui ponteiros nem gerenciamento manual de memória

Em linguagens como C++, composição forte pode ser implementada com ponteiros e destrutores que liberam explicitamente os objetos:

Mas em Java:

- Não há delete.
- O método finalize() foi **deprecado** e **não confiável**.
- O close() de AutoCloseable serve **para liberar recursos**, não memória.

Java Language Specification, Chapter 12 – https://docs.oracle.com/javase/specs/jls/se17/html/jls-12.html

JEP 421 – Deprecate Finalization for Removal – https://openjdk.org/jeps/421

[pdf do capítulo no github] Bloch, Joshua. *Effective Java*, 3rd Edition. Item 8: "Avoid finalizers and cleaners."