Tarefa por Fases — Interfaces em C# (equipes, repositório único, avaliação por equipe)

Introdução (para postar no ClassHero)

Objetivo geral. Consolidar a jornada **procedural → OO → interfaces → repository** com foco em design consciente (ISP, baixo acoplamento, testabilidade). O trabalho será **realizado em equipes e avaliado em equipes**, em **Fases** numeradas e encadeadas. Cada Fase tem um **enunciado** e uma **descrição** com orientações para execução.

Organização por Fases. Todas as entregas serão registradas como **Fase 0, Fase 1, Fase 2, ...** até a Fase final (mini-projeto). O(a) professor(a) fará a **distribuição de equipes para cada Fase** (as equipes podem variar entre fases). Cada Fase possui uma parte teórica, exemplos e uma **tarefa prática** que deve ser entregue no ClassHero com link/arquivo do repositório.

Repositório único (obrigatório). Cada equipe manterá **um repositório único** para todo o trabalho, com **projetos/pastas específicas por Fase**, garantindo rastreabilidade e continuidade.

Estrutura sugerida:

```
repo-raiz/
  README.md
  src/
    fase-00-aquecimento/
    fase-01-procedural/
    fase-02-oo-sem-interface/
    fase-03-com-interfaces/
    fase-04-repository-inmemory/
    fase-05-repository-csv/
    fase-06-repository-json/
    fase-07-isp/
    fase-08-testes-dubles/
    fase-09-cheiros-antidotos/
    fase-10-eixos-opcional/
    fase-11-mini-projeto/
  tests/
    fase-01/ ... (opcional; pode ficar junto de cada projeto)
  docs/
    arquitetura/    # diagramas/resumos curtos por Fase
    decisoes/       # ADRs curtos (opcional)
```

> Observação: o `README.md` na raiz **deve conter a composição da equipe (nomes e RAs)**, o sumário de Fases e instruções de execução/teste por Fase.

README (obrigatório e "vivo"). O `README.md` deve explicar **detalhadamente**: - **Composição da equipe** (nomes completos e RAs) — **item obrigatório**. - **Sumário** com links/âncoras para cada Fase. -

Como executar e testar cada Fase (comandos, projeto alvo, dependências, exemplos de uso). - **Decisões de design** por Fase (1–3 bullets: contratos, ISP, ponto de composição, trocas de implementação). - **Checklist de qualidade** aplicado (contratos coesos, alternância sem alterar cliente, testes sem I/O em unit, ausência de `switch`/downcasts, mudanças pequenas e localizadas). - **Evidências de testes** (prints curtos/logs mínimos quando fizer sentido).

Fluxo de entrega por Fase (simplificado). 1) A equipe cria a **pasta da Fase** no repositório e implementa o solicitado. 2) Atualiza o **README** com composição da equipe, como rodar/testar e as decisões de design da Fase. 3) Publica a entrega no ClassHero (link do repositório + notas da Fase).

Avaliação (por equipe). - **Correção e contrato**: funciona e alterna implementações sem mudar o cliente. - **Qualidade de design**: ISP/coesão, baixo acoplamento, classes folhas `sealed` quando aplicável, nomes claros. - **Testabilidade**: unit sem I/O; dublês adequados; cobertura dos caminhos críticos. - **Clareza de documentação**: README com composição da equipe, decisões de design e instruções de execução/teste. - **Evolução incremental**: mudanças focadas por Fase; histórico coerente e rastreável no repositório único.

Fases (Enunciados + Descrições — sem código)

Fase 0 — Orientações ao aluno (sem código)

Objetivo da Fase
Treinar o olhar de design: identificar **um objetivo fixo** e **peças alternáveis** que realizam esse objetivo por caminhos diferentes. Nomear o **contrato** (o que fazer) e duas **implementações** (como fazer), além de propor uma **política simples** para escolher entre as peças.

Exemplos de situações (para inspirar)
- **Pagamento**: contrato = “processar pagamento”; implementações = Pix / Cartão; política = “valores > R$ 500 → Cartão; senão → Pix”.
- **Envio de notificação**: contrato = “notificar usuário”; implementações = E-mail / SMS; política = “se usuário offline → SMS; se não → E-mail”.
- **Exportação de relatório**: contrato = “exportar relatório”; implementações = CSV / JSON; política = “para integração com planilha → CSV; para API → JSON”.
- **Tradução de texto**: contrato = “traduzir para EN”; implementações = Tradutor humano / API; política = “prazo curto → API; documento crítico → humano”.
- **Compressão de arquivo**: contrato = “comprimir”; implementações = ZIP / GZIP; política = “arquivos múltiplos → ZIP; único e grande → GZIP”.

O que é contrato aqui?
Uma frase curta que descreve **o que** a ação entrega, **sem** falar do “como”. Ex.: “enviar mensagem para o usuário” (não diga “via SMTP”... isso já é implementação).

Como escrever a política (simples e clara)
- **Forma 1 (regra)**: “Se *condição*, use *implementação A*; caso contrário, *implementação B*.”
- **Forma 2 (cenários)**: “No turno noturno → A; em urgência → B.”
- Evite “depende” genérico; sempre **amarre a escolha** a uma condição concreta (tempo, custo, confiabilidade, canal disponível).

Formato da entrega (2 casos reais)
Para **cada** caso, escreva **4–6 linhas**: 1) **Objetivo** (o que se quer alcançar)
2) **Contrato** (o que a peça deve fazer, em 1 linha)
3) **Implementação A** (como 1)
4) **Implementação B** (como 2)
5) **Política** (quando/como escolher A vs. B)
6) **Risco/Observação** (ex.: "SMS tem custo", "CSV perde tipos")

Entregar em texto simples (sem código). Título: "Fase 0 — [Nome da Equipe]".

Critérios de aceite (checklist rápido)
- Cada caso tem **objetivo, contrato, duas implementações e política** bem definida.
- O **contrato** não revela "como" (é descritivo, não técnico).
- As implementações são **alternáveis** (não são variações cosméticas da mesma coisa).
- A política é **concreta e aplicável** (não ambígua).
- Há pelo menos **um risco/observação** por caso.

Erros comuns a evitar
- Confundir contrato com implementação (ex.: "contrato = enviar e-mail").
- Trazer implementações idênticas (ex.: "JSON v1" vs. "JSON v2").
- Política vaga ("depende do contexto" sem dizer **qual** contexto).
- Casos abstratos demais; escolha situações **reais** do cotidiano ou do projeto.

Tempo sugerido
- **Em sala (dinâmico, equipes): 12–15 minutos** total
• 2–3 min para escolher os 2 casos
• 7–9 min para escrever objetivos/contratos/implementações/políticas
• 2–3 min para revisar com o checklist
- **Se quiser aprofundar (aula de 90 min):** estenda para **20 minutos** e peça **1 minuto de pitch** por equipe (apontando o contrato e a política).

Como o professor pode ajudar (perguntas gatilho)
- "O **contrato** está descrevendo o **que** (resultado) ou o **como** (técnica)?"
- "Essas duas implementações são realmente **alternáveis** para o mesmo objetivo?"
- "A **política** é acionável? Dá para uma pessoa terceira aplicá-la sem dúvida?"
- "Qual **risco** existe em cada implementação (custo, latência, erro) e isso muda a política?"

Rubrica enxuta (Fase 0)
- **Clareza do contrato** (0–3)
- **Alternância real entre implementações** (0–3)
- **Política objetiva e aplicável** (0–3)
- **Risco/observação pertinente** (0–1)
Total: 10 pts (sugestão para somar à rubrica geral)

Exemplo de Entrega — Fase 0 (Pagamento)
1) **Objetivo:** permitir que o cliente pague sua compra com sucesso, de forma segura e rápida.
2) **Contrato: processar pagamento** (recebe valor e dados do pagador, retorna sucesso/erro).
3) **Implementação A (como 1): Pix** — transferência instantânea via chave.
4) **Implementação B (como 2): Cartão** — autorização via operadora (crédito/débito).
5) **Política: valores > R$ 500 → Cartão; caso contrário → Pix**.

6) **Risco/Observação:** Pix pode falhar por indisponibilidade do banco; Cartão pode ter taxas/negações; documentar mensagens claras de erro ao cliente.

Fase 1 — Heurística antes do código (mapa mental)

Título para o ClassHero: Fase 1 — Heurística antes do código (mapa mental)

Descrição (para o ClassHero): Em equipes, produzam **uma página de mapa de evolução** para um problema trivial escolhido por vocês (ex.: formatar texto, notificar usuário, calcular frete). O mapa deve mostrar, em três quadros, **como o design evolui**: 1) **Versão procedural** — onde surgem `if/switch` e decisões de modo/variação. 2) **OO sem interface** — quem encapsula o quê; que partes continuam rígidas. 3) **Com interface** — **qual contrato** permite alternar implementações e **onde fica o ponto de composição** (injeção/fábrica). Incluam **3 sinais de alerta previstos** (ex.: interface gorda; downcasts; cliente mudando ao trocar implementação; testes lentos por I/O).

Entregáveis (no repositório único da equipe): - Pasta: `src/fase-01-procedural/` (pode conter apenas o artefato do mapa em `.md` ou imagem `.png/.pdf` nesta fase). - **README.md (raiz) atualizado** com: (a) **composição da equipe (nomes e RAs)**; (b) Sumário das Fases (link/âncora para a Fase 1); (c) onde encontrar e como visualizar o mapa (ex.: `docs/arquitetura/fase-01-mapa.*` ou dentro da pasta da fase).

Formato sugerido do mapa (modelo): - **Problema escolhido (1–2 linhas)** - **Quadro 1 — Procedural:** breve diagrama/fluxo + 2–3 bullets de "onde surgem if/switch". - **Quadro 2 — OO sem interface:** classes envolvidas + 2–3 bullets do que melhorou e do que ficou rígido. - **Quadro 3 — Com interface:** nome do **contrato** e **ponto de composição** + 2–3 bullets do que passa a variar sem tocar o cliente. - **Sinais de alerta (3 itens):** liste os cheiros que você **prevê** se não adotar o contrato/composição.

Critérios de avaliação (rubrica enxuta, 0–10): - **Cobertura dos 3 quadros** (procedural, OO, interface) — 0–3 - **Clareza do contrato e do ponto de composição** — 0–3 - **Qualidade dos 3 sinais de alerta previstos** — 0–2 - **Organização e documentação no README (composição da equipe, links)** — 0–2

Tempo sugerido em sala: 15–20 minutos (equipes) - 3–5 min: escolher o problema trivial - 8–10 min: construir os 3 quadros - 3–5 min: listar sinais de alerta e atualizar o README

Observações / pitfalls a evitar: - Não confundir **contrato** com classe concreta (o contrato descreve "o que", não "como"). - Evitem mapas genéricos: **ancorem em um problema concreto** (frete, formato de saída, notificação). - O **ponto de composição** deve ficar claro (onde as peças são escolhidas), não espalhado no cliente.

Peso sugerido: 10/100 — é a ponte conceitual que guiará as próximas fases com código.

Exemplo preenchido — Fase 1 (tema: Pagamento) - **Problema escolhido:** permitir que o cliente pague a compra escolhendo automaticamente o meio adequado (Pix ou Cartão) conforme regras simples de valor/risco. - **Quadro 1 — Procedural:** fluxo com `if (valor > 500) → Cartão; senão → Pix`; cada novo meio adiciona `if/switch`, aumentando casos e testes. - **Quadro 2 — OO sem interface:** `PixProcessor` e `CardProcessor` encapsulam o "como", mas o serviço ainda decide **qual** concreto usar; cliente muda quando trocamos meio. - **Quadro 3 — Com interface:** contrato = "processar pagamento"; implementações alternáveis; **ponto de composição** lê política "> 500 → Cartão; senão → Pix". **Efeito:** cliente **não muda** ao alternar

peças/política; testes ficam simples com dublês. - **Sinais de alerta (3):** (1) cliente muda ao trocar implementação; (2) ramificações espalhadas; (3) testes lentos/instáveis por I/O real.

Fase 2 — Procedural mínimo (ex.: formatar texto)

Título para o ClassHero: Fase 2 — Procedural mínimo (ex.: formatar texto)

Descrição (para o ClassHero): Em equipes, **implementem a ideia de "modos"** para um objetivo simples (ex.: formatar texto: upper/lower/title; escolher canal: email/sms/push). Devem existir **pelo menos 3 modos + 1 modo padrão**. Entreguem o **fluxo/descrição da função** e **5 cenários de teste/ fronteira descritos em texto**. Expliquem em poucas linhas **por que esse design procedural não escala** (acoplamento a `if/switch`, duplicação, dificuldade de extensão/teste).

Entregáveis (no repositório único da equipe): - Criem a **pasta da Fase 2** no repositório (seguir o padrão já adotado nas fases anteriores). - Um artefato **sem código** (`.md`, `.png` ou `.pdf`) contendo: - **Descrição do objetivo** e dos **modos** escolhidos (mín. 3 + padrão). - **Fluxo da função** (texto ou diagrama simples) destacando onde ocorrem os `if/switch`. - **5 cenários de teste/fronteira** descritos em texto (ver modelo abaixo). - **Notas de limitação**: por que essa abordagem não escala e onde doeria ao adicionar novos modos. - **README.md (raiz)** atualizado com: (a) **composição da equipe (nomes e RAs)**; (b) sumário com link para a Fase 2; (c) onde encontrar o artefato da Fase 2.

Formato sugerido (modelo de entrega em texto): - **Objetivo (1–2 linhas):** o que a função/fluxo pretende entregar. - **Modos (mín. 3 + padrão):** liste os nomes e o efeito esperado de cada modo; indique o **modo padrão**. - **Fluxo procedural (3–6 linhas):** descreva a sequência e **onde** as decisões (`if/switch`) ocorrem. - **5 cenários de teste/fronteira (apenas texto):** 1) valor/entrada mínima (ex.: string vazia) 2) valor/entrada máxima/limite (ex.: tamanho grande) 3) **modo inválido** (esperado cair no **padrão**) 4) combinação que revela ambiguidade entre modos (explicar decisão) 5) caso comum representativo do objetivo - **Por que não escala (4–6 linhas):** riscos claros (ex.: cada modo novo cria mais `if`, espalha decisão, testa combinações, dificulta leitura/manutenção).

Observação importante (C# ao final):

> **Sugestão didática**: ao final desta Fase, **incluir como referência** (no repositório) um pequeno `Program` **em C#** demonstrando a **versão puramente procedural** com `switch/if` (apenas para estudo). **Não é necessário entregar código nesta Fase no ClassHero** — a entrega oficial aqui é **conceitual/textual**.

Critérios de avaliação (rubrica enxuta, 0–10): - **Clareza do objetivo e definição dos modos** (3+) — 0–3 - **Fluxo procedural compreensível** (onde estão as decisões) — 0–3 - **Qualidade dos 5 cenários de teste/fronteira** — 0–2 - **Análise de limites ("por que não escala")** — 0–2

Tempo sugerido em sala: 15–20 minutos - 3–5 min: escolher objetivo e modos - 8–10 min: descrever fluxo e cenários de teste - 3–5 min: redigir limitações e atualizar o README

Pitfalls a evitar: - Modos que não mudam comportamento (variações cosméticas). - "Modo padrão" mal definido (deve ser inequívoco quando nada casar). - Cenários de teste genéricos demais (sem fronteiras/limites/ inválidos). - Crítica superficial ("não escala porque é feio"); foquem em **manutenção/ extensibilidade/testes**.

Peso sugerido: 10/100 — estabelece a dor do procedural e prepara a virada da Fase 3 (polimorfismo/OO sem interface).

Fase 3 — OO sem interface

Título para o ClassHero: Fase 3 — OO sem interface (herança + polimorfismo)

Descrição (para o ClassHero): Transformem a solução da Fase 2 em uma **hierarquia orientada a objetos** com **base comum** e **variações concretas**. A meta é **substituir decisões explícitas** do fluxo por **polimorfismo** (delegar o "como" às subclasses). Mantenham cada classe concreta **restrita à sua responsabilidade** e expliquem **o que melhorou** e **o que ainda ficou rígido** (cliente ainda conhece concretos? ponto de composição ausente?).

Entregáveis (no repositório único da equipe): - Pasta: `src/fase-02-oo-sem-interface/` (mantendo o padrão da estrutura sugerida). - Artefato de design **com pequenos trechos de código C#** (`.md` + snippets) contendo: - **Diagrama/descrição** da hierarquia (base + variações concretas). - **Trechos de código** mostrando a **classe base (virtual/abstract)** e **duas ou mais concretas**. - **Uso do cliente** (instanciação/seleção da concreta) e **onde o** `if/switch` **foi removido** do fluxo central. - **Análise "melhorou vs. rígido"** (2–3 bullets cada). - **README.md (raiz) atualizado** com: (a) **composição da equipe (nomes e RAs)**; (b) Sumário com link para a Fase 3; (c) onde encontrar e como executar o exemplo.

Exemplo mínimo (mesmo domínio da Fase 2: formatar texto)

```
// Base comum: descreve o ritual "formatar" e delega o "como" para as
concretas.
public abstract class TextFormatterBase
{
    public string Format(string text)
    {
        // Passos comuns poderiam ir aqui (ex.: normalização)
        return Apply(text);
    }
    protected abstract string Apply(string text); // passo variável
}

public sealed class UpperCaseFormatter : TextFormatterBase
{
    protected override string Apply(string text) => text?.ToUpperInvariant();
}

public sealed class LowerCaseFormatter : TextFormatterBase
{
    protected override string Apply(string text) => text?.ToLowerInvariant();
}

public sealed class TitleCaseFormatter : TextFormatterBase
{
    protected override string Apply(string text)
    {
```

```
        return System.Text.RegularExpressions.Regex.Replace(text ??
string.Empty,
            "\b(\p{L})", m => m.Value.ToUpperInvariant());
    }
}

// Cliente (ainda conhece concretos): melhora o fluxo, mas mantém a decisão
de composição aqui.
public static class FormatterClient
{
    public static string Render(string text, string mode)
    {
        TextFormatterBase fmt = mode switch
        {
            "UPPER" => new UpperCaseFormatter(),
            "lower" => new LowerCaseFormatter(),
            "Title" => new TitleCaseFormatter(),
            _ => new PassthroughFormatter()
        };
        return fmt.Format(text);
    }
}

public sealed class PassthroughFormatter : TextFormatterBase
{
    protected override string Apply(string text) => text; // padrão: mantém
}
```

> **Observação:** o `switch` acima **saiu do fluxo de formatação** (não há mais ramificações dentro do ritual). Ele permanece **apenas para compor** a concreta inicial. Na Fase 4, essa composição será extraída para um ponto único/contrato.

Como o polimorfismo substitui decisões - Antes: a função procedural **decidia** a cada chamada (vários `if/switch`). - Agora: o **fluxo comum** fica **linear** (classe base); o **passo variável** é **delegado** à concreta via `override`.

Responsabilidades (exemplo) - **TextFormatterBase**: orquestra o ritual (`Format`) e define o gancho `Apply`. - **Upper/Lower/Title/Passthrough**: implementam **somente** o passo variável. - **Cliente**: ainda **escolhe a concreta** (ponto de composição local — será endereçado na Fase 4).

Melhorou - Remoção de `if/switch` no **fluxo central**; leitura mais clara. - **Coesão por variação** (cada concreta trata seu "como"). - **Testes** de cada variação ficam pequenos e focados.

Ainda ficou rígido - **Cliente conhece concretos** (trocar/adição exige mexer no cliente). - **Composição dispersa** (decisão de seleção não é centralizada nem configurável). - **Difícil dobrar** em testes sem um contrato estável (prepara terreno para a Fase 4).

Critérios de avaliação (rubrica enxuta, 0–10) - **Hierarquia clara (base + concretas)** — 0–3 - **Substituição convincente de decisões por polimorfismo** — 0–3 - **Análise "melhorou vs. rígido"** — 0–2 - **README atualizado** (composição da equipe, sumário, execução) — 0–2

Tempo sugerido em sala: 15–20 minutos - 3–5 min: revisar a Fase 2 e definir base + variações - 8–10 min: escrever a base/concretas e o uso do cliente - 3–5 min: redigir análise e atualizar o README

Pitfalls a evitar - Base “faz-tudo” (mantenha apenas o que é realmente comum ao ritual). - Variações **cosméticas** (garanta comportamentos distintos e verificáveis). - Reintroduzir `if/switch` dentro das concretas (o passo variável deve ser simples e específico).

Peso sugerido: 10/100 — consolida a virada de “decidir” → “delegar” (polimorfismo) e prepara a introdução de contratos na próxima fase.

Fase 4 — Interface plugável e testável — Interface plugável e testável

Enunciado: Defina um **contrato** claro e refatore o cliente para depender dele.
Descrição: Explique como alternar implementações **sem mudar o cliente** e como dobrar a dependência em testes (injeção simples).

Fase 5 — Essenciais de interfaces em C

Enunciado: Proponha **duas interfaces** do seu domínio e uma classe que implementa **duas**.
Descrição: Explique quando usar **implementação explícita**, quando **genéricos com constraints** ajudam e quando **default members** devem ser evitados.

Fase 6 — ISP na prática (segregação por capacidade)

Enunciado: Dada uma interface “onipotente”, **segregue** em contratos coesos; ajuste o consumidor.
Descrição: Traga um “antes/depois” textual, indique **sinais de segregação** e ganhos obtidos.

Fase 7 — Repository progressivo (InMemory)

Enunciado: Modele um **repositório em memória** para seu domínio (ex.: itens, tarefas, filmes).
Descrição: Explique contrato, operações mínimas, e como um **serviço** usaria o repositório sem conhecer detalhes internos.

Fase 8 — Repository progressivo (CSV)

Enunciado: Evolua o repositório para **CSV**, mantendo o contrato.
Descrição: Descreva leitura/escrita, casos de arquivo inexistente e **por que o cliente não muda** ao alternar InMemory→CSV.

Fase 9 — Repository progressivo (JSON)

Enunciado: Evolua para **JSON**, preservando o contrato.
Descrição: Explique ida/volta, mutações e **ponto único de composição** para a troca CSV→JSON.

Fase 10 — Testabilidade: dublês e costuras

Enunciado: Planeje testes **sem I/O** usando dublês adequados.
Descrição: Liste cenários e resultados esperados, com foco em comportamento observável.

Fase 11 — Cheiros e antídotos (com *diffs* pequenos)

Enunciado: Identifique **dois cheiros** em fases anteriores e descreva a refatoração.
Descrição: Para cada cheiro, explique a motivação e o antídoto aplicado (ISP, composição, decisão centralizada).

Fase 12 (Opcional/Avançado) — Composição por eixos

Enunciado: Desenhe um **pipeline** com eixos independentes (ex.: filtrar → comprimir → armazenar) controlado por **política**.
Descrição: Mostre a ordem esperada das etapas, seleção por chave e como rejeitar combinações inválidas.

Fase 13 — Mini-projeto de consolidação (1 sprint curto)

Enunciado: Escolha **um domínio diferente** dos exemplos e entregue a solução completa por Fases, no mesmo repositório.
Descrição: Inclua uma **interface de uso mínima** (ex.: CLI), documentação de decisões e o checklist final (contratos coesos; troca sem mudar o cliente; testes rápidos; mudanças locais).