P1C3 - Organisez votre code avec MVC et les principes SOLID

Vérifiez votre travail

Votre code correspond maintenant à la **version intermédiaire** du dépôt. Comparez-le avec la proposition suivante.

Le test du bug. Il passe par l'API, comme les tests existants, car c'est le comportement observé par Leïla que vous voulez vérifier :

```
[Fact]
public async Task Balance_of_fixed_term_employee_counts_only_full_months()
{
    var client = _factory.CreateClient();

    var balance = await client.GetFromJsonAsync<double>("/api/leave/balance?employeeId=4");

    Assert.Equal(4.56, balance, precision: 2);
}
```

Sur le starter, il échoue avec la valeur 6,64 : c'est la preuve que le bug existe. Après le refactoring, il passe.

Les politiques et les constantes. Les règles des CDI et de l'alternance sont écrites une seule fois, avec des noms explicites :

```
public static class ContractTypes
{
    public const string Permanent = "CDI";
    public const string FixedTerm = "CDD";
    public const string Apprenticeship = "Alternance";
}

public sealed class PermanentContractPolicy : ILeaveEntitlementPolicy
{
    private const double AnnualDays = 25;

    public double ComputeEntitlement(Employee employee, DateTime today) => AnnualDays;
}

public sealed class ApprenticeshipPolicy : ILeaveEntitlementPolicy
{
    private const double AnnualDays = 25;
    private const double ExamPreparationDays = 5;

    public double ComputeEntitlement(Employee employee, DateTime today) =>
        AnnualDays + ExamPreparationDays;
}
```

Pour l'alternance, l'écriture `25 + 5` l'emporte sur `30` : les deux constantes nommées expliquent d'où vient le total. Les politiques des CDI et de l'alternance n'utilisent pas la date du jour, mais elles la reçoivent quand même : c'est le prix du contrat commun, et c'est ce qui permet au service de les traiter toutes de la même façon.

Le service. Les deux méthodes utilisent la même méthode `ComputeBalance` : la divergence est devenue impossible.

```
public enum ApprovalResult { Approved, InsufficientBalance, NotFound }

public sealed class LeaveService
{
    private readonly HamacDbContext _db;
    private readonly LeavePolicyResolver _policies;
    private readonly ILeaveNotifier _notifier;

    public LeaveService(HamacDbContext db, LeavePolicyResolver policies, ILeaveNotifier notifier)
    {
        _db = db;
        _policies = policies;
        _notifier = notifier;
    }

    public double? GetBalance(int employeeId)
    {
        var employee = _db.Employees.Find(employeeId);
        return employee is null ? null : ComputeBalance(employee);
    }

    public ApprovalResult Approve(int requestId)
    {
        // Filtrage en mémoire conservé : il sera corrigé au chapitre 4.
        var request = _db.LeaveRequests.ToList().FirstOrDefault(r => r.Id == requestId);
        if (request is null) return ApprovalResult.NotFound;

        var employee = _db.Employees.Find(request.EmployeeId)!;

        if (ComputeBalance(employee) < request.WorkingDays)
        {
            request.Status = "Refused";
            _db.SaveChanges();
            return ApprovalResult.InsufficientBalance;
        }

        request.Status = "Approved";
        employee.DaysTaken += request.WorkingDays;
        _db.SaveChanges();

        _notifier.Notify(employee, "Votre demande de congés est approuvée.");
        return ApprovalResult.Approved;
    }

    private double ComputeBalance(Employee employee) =>
        _policies.For(employee.ContractType).ComputeEntitlement(employee, DateTime.Today)
        - employee.DaysTaken;
}
```

Deux choix méritent une explication. `Approve` renvoie un `ApprovalResult`, et non un message ou un statut HTTP : le service décrit ce qui s'est passé dans les termes du métier, et c'est au contrôleur de le traduire en réponse. De même, `GetBalance` renvoie `null` pour un salarié introuvable, sans décider du message à afficher.

Le contrôleur. Il ne contient plus aucune règle RH ni aucun accès aux données :

```
[ApiController]
[Route("api/leave")]
public class LeaveController : ControllerBase
{
    private readonly LeaveService _leaveService;
```

```
    public LeaveController()
    {
        // Assemblage provisoire : il sera confié au conteneur .NET au chapitre 4.
        var policies = new LeavePolicyResolver(new Dictionary<string, ILeaveEntitlementPolicy>
        {
            [ContractTypes.Permanent] = new PermanentContractPolicy(),
            [ContractTypes.FixedTerm] = new FixedTermContractPolicy(),
            [ContractTypes.Apprenticeship] = new ApprenticeshipPolicy(),
        });

        _leaveService = new LeaveService(new HamacDbContext(), policies, new NotificationService());
    }

    [HttpGet("balance")]
    public IActionResult GetBalance(int employeeId)
    {
        var balance = _leaveService.GetBalance(employeeId);
        return balance is null ? Ok("Employee not found") : Ok(balance);
    }

    [HttpGet("approve")]
    public IActionResult Approve(int id) => _leaveService.Approve(id) switch
    {
        ApprovalResult.Approved => Ok("Approved"),
        ApprovalResult.InsufficientBalance => Ok("Refused: insufficient balance"),
        _ => Ok("Leave request not found"),
    };
}
```

Les méthodes d'action tiennent désormais en quelques lignes, et elles passent le test du contrôleur bien organisé : aucune règle métier, aucun accès direct aux données. Le constructeur, lui, connaît encore toutes les classes concrètes : c'est l'assemblage provisoire annoncé, que vous supprimerez au prochain chapitre.

Le cas d'une demande introuvable faisait planter le starter, car `req` valait `null`. Il renvoie désormais un message : c'est le seul autre changement de comportement, et il ne casse aucun appel existant.

Le test unitaire. Il est désormais possible de tester une règle RH sans lancer l'API ni la base de données :

```
[Fact]
public void Fixed_term_policy_counts_only_full_months()
{
    var policy = new FixedTermContractPolicy();
    var employee = new Employee { HireDate = new DateTime(2026, 3, 15) };

    var entitlement = policy.ComputeEntitlement(employee, new DateTime(2026, 10, 20));

    Assert.Equal(14.56, entitlement, precision: 2);
}
```

Ce test s'exécute en quelques millisecondes et donnera le même résultat dans un an, grâce à la date passée en paramètre. Vous écrirez le même type de test pour la politique des stages.

La distinction attendue est la suivante : un code **déplacé** se retrouve dans une autre classe, avec les mêmes défauts ; un code **réorganisé** a changé de structure, de sorte que la règle n'existe plus qu'à un seul endroit et peut être testée seule. C'est ce second résultat qu'Inès attendait.

Erreurs fréquentes

- **Créer les collaborateurs dans le service** : si votre `LeaveService` crée lui-même ses politiques ou son notificateur avec `new`, vous avez déplacé le couplage au lieu de le réduire. Seul le point d'assemblage, provisoirement le contrôleur, doit connaître les classes concrètes.
- **Lire `DateTime.Today` dans une politique** : votre test unitaire passera aujourd'hui, puis échouera dans quelques semaines.
- **Modifier les routes ou les messages de l'API** : les tests existants échouent, et vous ne savez plus si c'est à cause du refactoring ou du changement de contrat.
- **Corriger le filtrage après `ToList()` en même temps** : ce n'est pas faux, mais vous mélangez deux changements. Gardez-le pour le chapitre 4.
- **Renvoyer `-1` ou lever une exception dans une politique** pour signifier « pas de droits » : le service ne peut plus faire confiance au contrat.