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 :

```
@Test
void balanceOfFixedTermEmployeeCountsOnlyFullMonths() throws Exception {
    var result = mvc.get().uri("/api/leave/balance?employeeId=4").exchange();

    var balance = Double.parseDouble(result.getResponse().getContentAsString());
    assertThat(balance).isCloseTo(4.56, within(0.01));
}
```

Sur le starter, il échoue avec une valeur d'environ 6,64 : c'est la preuve que le bug existe. Après le refactoring, il passe. La comparaison se fait avec une tolérance, `within(0.01)`, car un calcul en `double` comme `7 * 2.08` ne tombe pas toujours exactement sur 14,56.

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

```
public final class PermanentContractPolicy implements LeaveEntitlementPolicy {

    private static final double ANNUAL_DAYS = 25;

    @Override
    public boolean supports(ContractType type) {
        return type == ContractType.PERMANENT;
    }

    @Override
    public double computeEntitlement(Employee employee, LocalDate today) {
        return ANNUAL_DAYS;
    }
}

public final class ApprenticeshipPolicy implements LeaveEntitlementPolicy {

    private static final double ANNUAL_DAYS = 25;
    private static final double EXAM_PREPARATION_DAYS = 5;

    @Override
    public boolean supports(ContractType type) {
        return type == ContractType.APPRENTICESHIP;
    }

    @Override
    public double computeEntitlement(Employee employee, LocalDate today) {
        return ANNUAL_DAYS + EXAM_PREPARATION_DAYS;
    }
```

```
}
```

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, INSUFFICIENT_BALANCE, NOT_FOUND }

public final class LeaveService {

    private final EmployeeRepository employeeRepository;
    private final LeaveRequestRepository leaveRequestRepository;
    private final LeavePolicyResolver policies;
    private final LeaveNotifier notifier;

    public LeaveService(EmployeeRepository employeeRepository,
                        LeaveRequestRepository leaveRequestRepository,
                        LeavePolicyResolver policies,
                        LeaveNotifier notifier) {
        this.employeeRepository = employeeRepository;
        this.leaveRequestRepository = leaveRequestRepository;
        this.policies = policies;
        this.notifier = notifier;
    }

    public Optional<Double> getBalance(long employeeId) {
        return employeeRepository.findById(employeeId).map(this::computeBalance);
    }

    public ApprovalResult approve(long requestId) {
        // Filtrage en mémoire conservé : il sera corrigé au chapitre 4.
        var found = leaveRequestRepository.findAll().stream()
                .filter(r -> r.getId() == requestId)
                .findFirst();
        if (found.isEmpty()) {
            return ApprovalResult.NOT_FOUND;
        }
        var request = found.get();
        var employee = employeeRepository.findById(request.getEmployeeId()).orElseThrow();

        if (computeBalance(employee) < request.getWorkingDays()) {
            request.setStatus("REFUSED");
            leaveRequestRepository.save(request);
            return ApprovalResult.INSUFFICIENT_BALANCE;
        }

        request.setStatus("APPROVED");
        employee.setDaysTaken(employee.getDaysTaken() + request.getWorkingDays());
        leaveRequestRepository.save(request);
        employeeRepository.save(employee);

        notifier.notify(employee, "Votre demande de congés est approuvée.");
        return ApprovalResult.APPROVED;
    }

    private double computeBalance(Employee employee) {
        var type = ContractType.fromCode(employee.getContractType());
        return policies.policyFor(type).computeEntitlement(employee, LocalDate.now())
                - employee.getDaysTaken();
```

```
    }
}
```

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 un `Optional` vide pour un salarié introuvable, sans décider du message à afficher. Le service ne renvoie jamais `null` : l'absence de résultat est visible dans sa signature.

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

```
@RestController
@RequestMapping("/api/leave")
public class LeaveController {

    private final LeaveService leaveService;

    public LeaveController(EmployeeRepository employeeRepository,
                           LeaveRequestRepository leaveRequestRepository) {
        // Assemblage provisoire : il sera confié au conteneur Spring au chapitre 4.
        var policies = new LeavePolicyResolver(List.of(
                new PermanentContractPolicy(),
                new FixedTermContractPolicy(),
                new ApprenticeshipPolicy()));
        this.leaveService = new LeaveService(employeeRepository, leaveRequestRepository,
                policies, new NotificationService());
    }

    @GetMapping("/balance")
    public Object getBalance(@RequestParam long employeeId) {
        // Contrat de l'API inchangé jusqu'au chapitre 5.
        var balance = leaveService.getBalance(employeeId);
        return balance.isPresent() ? balance.get() : "Employee not found";
    }

    @GetMapping("/approve")
    public String approve(@RequestParam long id) {
        return switch (leaveService.approve(id)) {
            case APPROVED -> "Approved";
            case INSUFFICIENT_BALANCE -> "Refused: insufficient balance";
            case NOT_FOUND -> "Leave request not found";
        };
    }
}
```

Les méthodes tiennent désormais en quelques lignes, et elles passent le test du contrôleur bien organisé : aucune règle métier, aucun accès direct aux données. Vous retrouvez le `switch` exhaustif : si un quatrième résultat apparaît un jour dans `ApprovalResult`, le contrôleur ne compilera plus tant qu'il ne l'aura pas traité. 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, avec une `NullPointerException`. 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 démarrer Spring ni la base de données. Placez-le dans `src/test/java/com/hamac/service/FixedTermContractPolicyTests.java` :

```
class FixedTermContractPolicyTests {
```

```
    @Test
    void countsOnlyFullMonths() {
        var policy = new FixedTermContractPolicy();
        var employee = new Employee();
        employee.setHireDate(LocalDate.of(2026, 3, 15));

        var entitlement = policy.computeEntitlement(employee, LocalDate.of(2026, 10, 20));

        assertThat(entitlement).isCloseTo(14.56, within(0.01));
    }
}
```

Cette classe ne porte aucune annotation Spring : c'est un simple test JUnit. Il 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 `LocalDate.now()` dans une politique** : votre test unitaire passera aujourd'hui, puis échouera dans quelques semaines.
- **Annoter dès maintenant vos classes avec `@Service` ou `@Component`** : Spring créerait ses propres instances, en plus de celles que le contrôleur crée avec `new`. Vous ferez cette transition proprement au chapitre 4.
- **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 `findAll()` 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.