1. Expérience et Rôle dans le Projet

Mon Rôle :

Mon travail s'est concentré sur la représentation des personnes, des étages, et de leur comportement dans le bâtiment.

Compétences Développées :

- Programmation orientée objet avancée en Java
- Utilisation des **Records Java** (Habitude)
- Gestion du temps avec **LocalTime**
- Design d'interfaces et implémentation
- Gestion de l'immutabilité et copies défensives
- Enums avec méthodes personnalisées
- Validation de données et gestion d'erreurs

2. Travail Réalisé

2.1 Interface Personne

Conception de l'interface principale définissant le contrat pour toute personne dans le système :

Points clés :

- Séparation claire entre état et comportement
- Méthodes par défaut pour éviter la duplication de code
- Intégration avec le système d'appels d'ascenseur (`Call`)

2.2 Classe PersonneSimple

Implémentation complète d'une personne avec suivi d'état de déplacement :

Attributs gérés :

- `id` : Identifiant unique
- `etage` : Position actuelle
- `habitudes` : Liste immuable d'habitudes
- `enDeplacement` : État booléen de déplacement
- `etageDestination` : Destination si en déplacement

2.3 Enum Horaire

Gestion des plages horaires de la journée avec logique de comparaison

2.4 Enum Event

Classification des types d'événements :

- `ENTREE_BATIMENT` / `SORTIE_BATIMENT`
- `INVITATION_ARRIVEE` / `INVITATION_DEPART`
- `DEPLACEMENT_INTERNE`

2.5 Enum Journee

Types de journées : `SEMAINE`, `WEEKEND`, `FERIE`

2.7 Classe Etage

Gestion complète d'un étage du bâtiment :

Attributs :

- `niveau` : Numéro d'étage
- `nbHabitants` : Capacité maximale
- `personnes` : Liste des personnes présentes

3. Difficultés Rencontrées

3.1 Synchronisation de l'État de Déplacement

Problème : Risque d'incohérence entre l'état « enDeplacement » et la position réelle de la personne.

Cas problématiques :

// Personne en déplacement mais arrivée à destination

personne.deplacerVers(5); // enDeplacement = true, destination = 5

ascenseur.depose(personne); // étage = 5

// enDeplacement toujours à true !

Conséquences :

- Personnes bloquées en état de déplacement

- Impossible de créer de nouveaux déplacements

- Statistiques faussées

3.2 Immutabilité et Sécurité des Collections

Problème : Protection contre les modifications externes des listes.

Risque identifié :

List<Habitude> habitudes = personne.habitudes();

habitudes.clear(); // Modifie l'état interne !

Enjeux :

- Violation du principe d'encapsulation

- Bugs difficiles à tracer

- État corrompu de l'objet

3.3 Validation des Données

Problème : Garantir la cohérence des données dès la construction.

Points de vigilance :

- Étages négatifs

- Destinations invalides

- Heures nulles

- Capacité d'étage dépassée

4. Solutions Apportées

4.1 Solution pour la Synchronisation des États

Approche : Réinitialisation automatique lors de l'arrivée à destination.

```
@Override
public void setEtage(int nouvelEtage) {
   this.etage = nouvelEtage;

   // Vérification automatique de l'arrivée
   if (etageDestination != null && etageDestination == nouvelEtage) {
      this.enDeplacement = false;
      this.etageDestination = null;
   }
}
```

Avantages :
- Pas d'oubli possible
- Cohérence garantie
- Code centralisé
- Évite la duplication

4.2 Solution pour l'Immutabilité

```
public PersonneSimple(int id, int etage, List<Habitude> habitudes) {
   this.id = id;
   this.etage = etage;
   // Copie immuable : impossibilité de modifier de l'extérieur
   this.habitudes = List.copyOf(habitudes);
   this.enDeplacement = false;
   this.etageDestination = null;
}
```

Dans les getters (copie de retour) :

```
// Dans Etage
public List<Personnes> personnes() {
   return new ArrayList<>(personnes); // Nouvelle liste
}
```

Résultat :Protection complète contre les modifications externes.

4.3 Solution pour la Validation

```
public record Habitude(...) {
   public Habitude {
      if (debut == null || fin == null || type == null) {
         throw new IllegalArgumentException("Les champs ne peuvent pas être nuls");
      }
      if (destination < 0) {
         throw new IllegalArgumentException("La destination ne peut pas être négative");
      }
   }
}
```

```
public Etage(int niveau, int nbHabitants, List<Personnes> personnes) {
   if (niveau < 0) {
      throw new IllegalArgumentException("Le niveau d'un étage ne peut être négatif.");
   }
   if (nbHabitants < 0) {
      throw new IllegalArgumentException("Le nombre d'habitants ne peut être négatif.");
   }
   // ... initialisation
}
```

5. Points d'Amélioration Identifiés

1. Tests unitaires manquants
   - Transitions d'état dans PersonneSimple
   - Dépassement de capacité des étages

2. Documentation incomplète :
   - Manque de `@param`, `@return`, `@throws`
   - Exemples d'utilisation absents

3. Gestion des erreurs à renforcer :
   - Que se passe-t-il si on déplace une personne déjà en déplacement ?
   - Comportement si destination == étage actuel ?