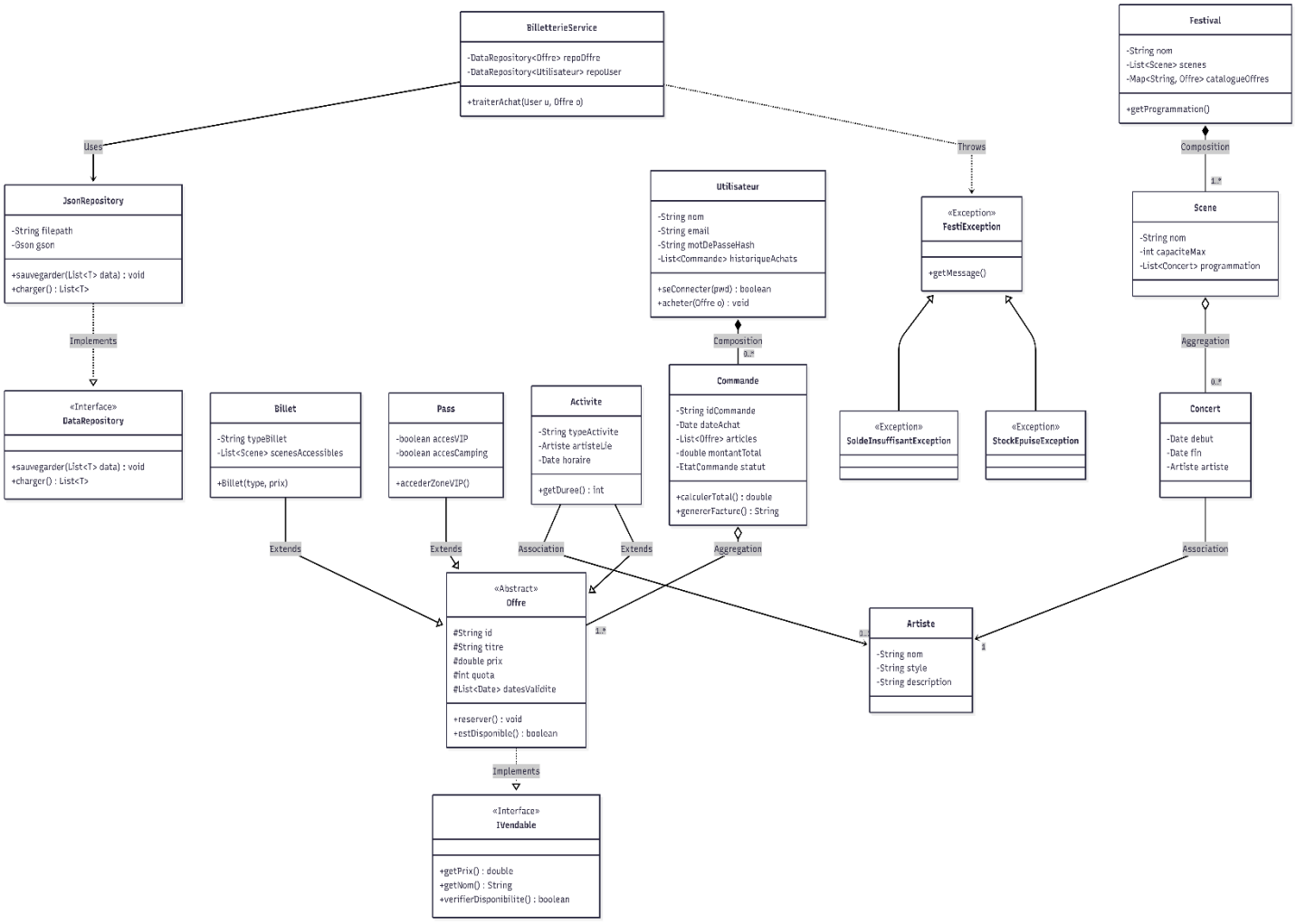

Justification des Choix Architecturaux et Techniques

Pour répondre aux exigences du cahier des charges FestiCore, nous avons conçu une architecture modulaire, robuste et extensible. Voici l'explication détaillée de nos décisions de conception :

3.1. Modélisation des Offres : Héritage et Polymorphisme

Pour gérer la diversité des produits vendus (Billets journée, Pass VIP, Activités annexes), nous avons refusé de créer des classes isolées.

- **Choix :** Création d'une classe abstraite Offre regroupant les attributs communs (prix, quota, id).
- **Justification :** Cela permet une factorisation du code. Grâce au **polymorphisme**, une Commande peut contenir une simple List<Offre> mélangeant indifféremment des billets et des activités, simplifiant ainsi le calcul du total et l'affichage du panier.

3.2. Persistance des Données : Utilisation de la Généricité

La sauvegarde des données (Utilisateurs, Réservations, Stocks) dans des fichiers JSON est une contrainte forte.

- **Choix :** Implémentation d'une interface générique DataRepository<T> et de sa classe concrète JsonRepository<T>.
- **Justification :** Plutôt que d'écrire un gestionnaire de fichier pour chaque type de données, cette classe générique permet de manipuler n'importe quelle entité du projet. Une seule implémentation suffit pour sauvegarder une List<Utilisateur> ou une List<Offre>, ce qui réduit considérablement le risque d'erreurs et la duplication de code.

3.3. Flexibilité du système : Interface IVendable

- **Choix :** Utilisation de l'interface IVendable définissant le contrat getPrix() et verifierDisponibilite().
- **Justification :** Cette abstraction permet de découpler la logique de vente de l'objet vendu. Si le festival décide demain de vendre du merchandising (T-shirts, goodies), il suffira que la nouvelle classe implémente IVendable pour être immédiatement compatible avec le système de facturation existant, sans modifier le cœur de l'application.

3.4. Structure du Festival : Composition et Collections

- **Choix :** Le Festival est composé d'une liste de Scene, qui elle-même agrège une liste de Concert.
- **Justification :** Nous avons utilisé la **composition** pour refléter la hiérarchie réelle : une scène n'a pas de sens sans le festival. Pour le stockage, nous avons privilégié les ArrayList pour les listes ordonnées (programmation) et les HashMap pour le catalogue des offres, permettant un accès instantané (complexité O(1)) à un produit via son ID lors d'un achat.

3.5. Robustesse : Gestion des Exceptions Métier

- **Choix :** Création d'une hiérarchie d'exceptions personnalisées héritant de FestiException (ex: StockEpuiseException, SoldeInsuffisantException).
- **Justification :** Cela permet de séparer clairement les erreurs techniques (bug, fichier manquant) des erreurs "métier" (plus de places). Le contrôleur peut ainsi capturer spécifiquement une StockEpuiseException pour proposer une alternative à l'utilisateur, au lieu de faire planter le programme.

3.6. Architecture en Couches (Separation of Concerns)

- **Choix :** Séparation stricte entre le modèle (model), l'accès aux données (dao) et la logique métier (service).
- **Justification :** Cette structure respecte les bonnes pratiques de génie logiciel. La classe Utilisateur ne sait pas comment s'enregistrer dans un fichier ; c'est le rôle du JsonRepository. Cela facilite les tests unitaires et la maintenance future.