Rapport de Conception - Projet StarFleetReservation

1. Introduction

Le projet **StarFleetReservation** est une application Java conçue pour gérer les réservations de missions spatiales dans l'univers fictif de *Starfleet*. Ce système permet de gérer les vaisseaux spatiaux, les missions, les réservations, et les personnes impliquées (officiers et civils), tout en offrant des fonctionnalités de sauvegarde et de chargement des données. Ce rapport explique les choix de conception effectués dans la structure, les classes, et les fonctionnalités, en lien avec les objectifs définis dans la documentation.

---

2. Objectifs du système

Les objectifs principaux du système sont :

1. Gérer les vaisseaux spatiaux (ajout, liste).
2. Planifier et gérer les missions spatiales.
3. Gérer les réservations pour les missions.
4. Gérer les personnes impliquées (officiers, civils).
5. Sauvegarder et charger les données.

Ces objectifs ont guidé les choix architecturaux et techniques décrits ci-dessous.

---

3. Choix d'architecture : MVC simplifié

Justification

J'ai opté pour une architecture **MVC simplifiée** (Modèle-Vue-Contrôleur) pour structurer le projet, comme mentionné dans la documentation. Voici pourquoi :

- **Modularité** : La séparation entre les données (Modèle), l'interface utilisateur (Vue), et la logique métier (Contrôleur) facilite la maintenance et les futures évolutions (ex. : ajout d'une GUI).
- **Simplicité** : Une version simplifiée du MVC convient à une application console de petite échelle, évitant une complexité inutile.
- **Réutilisabilité** : Les composants du Modèle peuvent être réutilisés indépendamment de la Vue ou du Contrôleur.

Implémentation

- **Modèle** : Les packages fr.starfleet.modele.* contiennent les entités (Vaisseau, Mission, Personne, Reservation), qui représentent les données du système.
- **Vue** : La classe InterfaceConsole dans fr.starfleet.ui gère l'interaction avec l'utilisateur via une interface en ligne de commande.
- **Contrôleur** : La classe SystemeReservation dans fr.starfleet.modele.systeme centralise la logique métier, comme la création de missions ou la gestion des réservations.

---

4. Conception des composants du Modèle

4.1 Classes du Modèle

Vaisseau

- **Choix** : Attributs nom, immatriculation, capaciteMaximale, et une liste de missions.
- **Justification** :
    - Le nom et l'immatriculation identifient un vaisseau de manière unique et réaliste dans un contexte spatial.
    - La capaciteMaximale reflète une contrainte physique essentielle pour limiter les réservations.
    - La liste de missions établit une relation bidirectionnelle avec Mission, permettant de suivre les affectations d'un vaisseau.
- **Validation** : Les setters incluent des vérifications (non-nullité, non-vide, capacité positive) pour garantir l'intégrité des données.

Mission

- **Choix** : Attributs code, description, dateDepart, dateRetour, destination, vaisseau, capaciteMaximale, et une liste de reservations.

- **Justification** :
    - Le code sert d'identifiant unique, essentiel pour associer des réservations.
    - Les dates (dateDepart, dateRetour) et la destination définissent la planification et l'objectif de la mission, alignés avec un scénario spatial.
    - La référence au vaisseau lie la mission à une ressource physique, tandis que capaciteMaximale limite les réservations possibles.
    - La liste de reservations permet de gérer les passagers de manière dynamique.
- **Méthodes** : ajouterReservation et annulerReservation contrôlent les places disponibles, renforçant la logique métier.

Personne (abstraite), Civil, Officier

- **Choix** : Une classe abstraite Personne avec des sous-classes Civil et Officier, utilisant l'héritage et une méthode abstraite getDescription.
- **Justification** :
    - L'héritage permet de factoriser les attributs communs (nom, prenom, identifiant) tout en spécialisant les rôles (officiers avec rang et specialite, civils avec planeteOrigine et motifVoyage).
    - La méthode getDescription offre une flexibilité pour afficher des informations spécifiques à chaque type de personne, améliorant la lisibilité dans l'interface.
- **Serializable** : Toutes les classes implémentent Serializable pour permettre la sauvegarde.

Reservation

- **Choix** : Attributs idReservation, passager, mission, dateReservation, confirmee.
- **Justification** :
    - L'idReservation est un identifiant unique généré automatiquement (ex. : "RES1"), essentiel pour suivre et annuler des réservations.
    - Les références à passager (Personne) et mission établissent les relations clés du système.
    - Le booléen confirmee permet de gérer l'état de la réservation (confirmée ou annulée), offrant une flexibilité dans la gestion.

4.2 Relations entre classes

- **Vaisseau ↔ Mission** : Une relation bidirectionnelle (un vaisseau a plusieurs missions, une mission est associée à un vaisseau).

- **Mission ↔ Reservation** : Une mission contient une liste de réservations, chaque réservation référence une mission.
- **Personne ↔ Reservation** : Une personne peut avoir plusieurs réservations (relation implicite via la liste globale dans SystemeReservation).

Ces choix reflètent une modélisation orientée objet claire et adaptée à un système de réservation.

---

5. Conception du Contrôleur : SystemeReservation

Choix

- Une classe centrale qui gère les listes (vaisseaux, missions, personnes, reservations) et les opérations principales (ajout, réservation, sauvegarde/chargement).
- Utilisation de ArrayList pour les collections et de Serializable pour la persistance.

Justification

- **Centralisation** : Regrouper la logique dans une seule classe simplifie la coordination entre les entités et évite la duplication de code.
- **Flexibilité** : Les méthodes comme effectuerReservation utilisent des streams Java pour rechercher des objets, rendant le code concis et performant.
- **Persistance** : La sérialisation Java est choisie pour sa simplicité et son intégration native, adaptée à un prototype console.

Limitations et améliorations

- Les méthodes sauvegarderDonnees et chargerDonnees actuelles affichent des messages mais ne permettent pas de passer un nom de fichier personnalisé ni de renvoyer le système chargé. Cela pourrait être ajusté pour correspondre à InterfaceConsole.

---

6. Conception de la Vue : InterfaceConsole

Choix

- Une interface en ligne de commande avec un menu interactif basé sur Scanner et des sous-menus pour chaque fonctionnalité (vaisseaux, personnes, missions, réservations).

Justification

- **Simplicité** : Une interface console est rapide à développer et suffisante pour un prototype fonctionnel, alignée avec les ressources et le temps disponibles.
- **Interactivité** : Les sous-menus (ex. : "1. Ajouter un vaisseau", "2. Lister les vaisseaux") rendent l'utilisation intuitive et structurée.
- **Robustesse** : Les entrées utilisateur sont gérées avec des vérifications minimales (ex. : parsing d'entiers), bien que des améliorations soient possibles (gestion d'exceptions).

Intégration avec le scénario

Le scénario de test (création de vaisseaux, ajout de personnes, etc.) a été implémenté directement dans InterfaceConsole, avec des appels aux méthodes de SystemeReservation, assurant une exécution fluide.

---

7. Gestion de la persistance

Choix

- Utilisation de la sérialisation Java pour sauvegarder l'état complet de SystemeReservation dans un fichier (sauvegarde.dat).

Justification

- **Efficacité** : La sérialisation permet de sauvegarder et restaurer toutes les données (y compris les relations entre objets) en une seule opération.
- **Simplicité** : Pas besoin d'une base de données ou d'un format complexe comme JSON pour un projet de cette échelle.
- **Conformité** : Aligné avec la documentation qui mentionne la sérialisation comme option.

Limites

- La méthode chargerDonnees actuelle ne met pas à jour l'instance courante (this), ce qui nécessite une modification pour fonctionner avec InterfaceConsole.

---

8. Utilitaires : DateUtil et FileUtil

Choix

- Ajout de DateUtil pour gérer les dates (formatage, parsing, validation).
- Ajout de FileUtil pour des opérations fichier complémentaires (lecture/écriture texte, vérification d'existence).

Justification

- **DateUtil** : Les dates dans Mission et Reservation nécessitent une gestion cohérente (ex. : parsing d'entrées utilisateur comme "01/04/2025"), rendant cet utilitaire essentiel.
- **FileUtil** : Bien que la sérialisation soit utilisée, FileUtil offre une flexibilité future (ex. : sauvegarde en texte clair) et des vérifications pratiques (ex. : fichier existant).

Problèmes rencontrés

- DateUtil ne générait pas de .class dans certains cas, probablement à cause d'une redirection par VS Code. Cela a été résolu en spécifiant explicitement les répertoires de sortie (-d).

---

9. Réponse au scénario de test

Le scénario de test fourni a été intégré dans la conception :

- **Création de vaisseaux/missions/personnes** : Implémentée via des sous-menus dans InterfaceConsole.
- **Réservations** : Gérées avec effectuerReservation et une logique de confirmation/annulation.
- **Listes de passagers** : Ajoutée comme option dans gererMissions pour afficher les réservations par mission.

- **Sauvegarde/chargement** : Fonctionnalité clé testée pour garantir la persistance.

---

10. Points forts et limites

Points forts

Modularité : La structure en packages (modele, ui, util) et l'utilisation de l'héritage rendent le code clair et extensible.

Robustesse : Les vérifications dans les setters et méthodes empêchent des états invalides.

Fonctionnalité complète : Le scénario de test est entièrement réalisable avec le code actuel.

Limites

Gestion des erreurs : Les exceptions (ex. : parsing invalide dans Scanner) ne sont pas pleinement gérées.

Persistance : La sérialisation est fragile face aux changements de structure des classes.

Interface : La console est limitée par rapport à une GUI.

---

11. Améliorations proposées

Conformément à la documentation :

- GUI : Remplacer InterfaceConsole par une interface graphique (ex. : JavaFX).
- Base de données : Utiliser SQLite ou une autre DB pour une persistance plus robuste.
- Tests unitaires : Ajouter JUnit pour valider chaque méthode (ex. : ajouterReservation).
- Équipages : Étendre Mission pour gérer des officiers assignés en plus des réservations.

---

12. Conclusion

Les choix de conception pour **StarFleetReservation** privilégient la simplicité, la modularité, et la fonctionnalité pour répondre aux objectifs initiaux. L'architecture MVC simplifiée, la modélisation orientée objet, et l'utilisation de la sérialisation offrent une base solide pour un prototype console, tout en laissant place à des améliorations futures. Le scénario de test valide la pertinence de ces décisions, démontrant un système opérationnel et cohérent.