Année 2024 -2025

PROJET MVC

SUP GALILEE - INFO3

TRAVAUX PRATIQUES
DEVOIR ET PROJET

MOTIFS ET PATRONS
DE CONCEPTION

Travail réalisé par :

- Jean-Yves BUISSERETH
- Ihssane ER-RAMI
- Arthur LE GOFF
- Ilyes SMAOUI

Travail encadré par :

- Guillaume SANTINI

Table des matières

Synthese du projet

Contexte et Objectifs du Projet

Le but de notre projet est de réaliser une application de gestion et utilisation d'un cinéma qui gère deux types d'utilisateurs
- Les Clients : Ils peuvent consulter les films disponibles, réserver des séances, visualiser le récapitulatif de leur réservation et, dans une version future, annuler une réservation.
- Le Manager : il a accès à des fonctionnalités de gestion comme la création de films et de séances, qui nécessitent des droits administratifs.

Composition de l'équipe INFO3

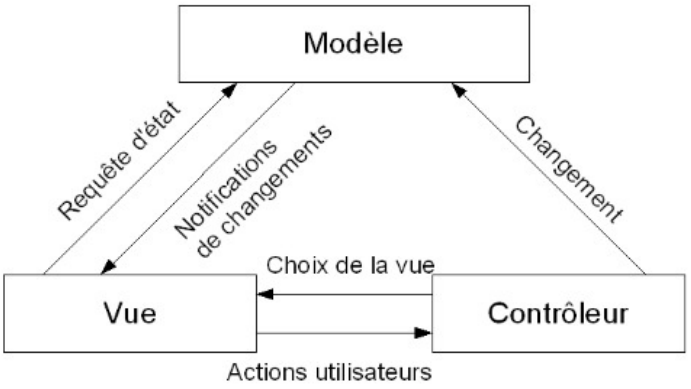
- Jean-Yves BUISSERETH
- Ihssane ER-RAMI
- Arthur LE GOFF
- Ilyes SMAOUI

Diagrammes

Diagramme scénarios

Acteurs et composants

- **Vue** : Affiche les interfaces (formulaires, menus, messages d'erreur ou de confirmation).
- **Contrôleur** : Ordonne les interactions entre la Vue et le Modèle en fonction des actions utilisateur.
- **Modèle** : Contient la logique métier (gestion des réservations, annulations, vérification des permissions, etc.).
    - **Client** : Peut consulter les films, infos sur les salles, infos sur les séances, réserver ou refaire des réservations.
    - **Manager** : Dispose des droits du Client, et peut également créer des films, des salles et des séances.



Scénario détaillé coté Client

1. **Arrivée au cinéma**
    - **Vue** : Affiche la page d'accueil.
2. **Connexion en tant que Client**
    - **Vue** : Affiche le formulaire de connexion.
    - **Contrôleur** : Reçoit et transmet les identifiants au Modèle pour vérification.
    - **Modèle** : Vérifie l'authentification et les droits d'accès.
    - **Contrôleur** : Selon le résultat, oriente vers le menu client ou affiche un message d'erreur.
3. **Affichage du menu client**
    - **Vue** : Affiche le menu proposant différentes actions (consulter les films, réserver, annuler une réservation, etc.).
    - **Remarque** : Ce menu est la porte d'entrée vers toutes les actions, et une fois connecté, le client peut enchaîner plusieurs actions (réservation, annulation, nouvelle réservation, etc.) jusqu'à sa déconnexion.
4. **Actions non autorisées** (cas d'erreurs)
    - **4a. Création d'un film**
        - **Vue** : L'utilisateur clique sur « Créer un film ».
        - **Contrôleur** : Transmet la demande au Modèle.
        - **Modèle** : Vérifie les permissions et rejette l'opération (cette action est réservée aux administrateurs).
        - **Contrôleur** : Retourne au menu en indiquant à la Vue d'afficher un message d'erreur ("Vous n'avez pas les permissions pour créer un fichier").
    - **4b. Création d'une séance**
        - **Vue** : L'utilisateur clique sur « Créer une séance ».
        - **Contrôleur** : Transmet la demande au Modèle.
        - **Modèle** : Rejette l'opération pour cause de permissions insuffisantes.
        - **Contrôleur** : Retourne au menu en affichant un message d'erreur correspondant.

4. **Actions autorisées** (cas de validation)

    - **4c. Consultation des disponibilités de films**
        - **Vue** : L'utilisateur sélectionne l'option "Voir les films disponibles".
        - **Contrôleur** : Demande au Modèle la liste des films et disponibilités.
        - **Modèle** : Renvoie les informations.
        - **Contrôleur** : Transmet ces données à la Vue qui les affiche.
5. **Choix d'un film pour réservation**
    - **Vue** : Permet au Client de sélectionner un film à réserver.
    - **Contrôleur** : Enregistre le choix et prépare le contexte de la réservation dans le Modèle.
6. **Initiation de la réservation**
    - **Vue** : Affiche l'interface de réservation (formulaire).
    - **Contrôleur** : Vérifie que le film a bien été sélectionné et transmet la demande au Modèle.
    - **Modèle** : Crée un objet "Réservation" dans le contexte en attente de complétion.
7. **Choix du nombre de billets et des types**
    - **Vue** : Affiche le formulaire permettant de choisir le nombre de billets et les types (par exemple : billet -16 ans, billet Senior, etc.).
    - **Contrôleur** : Récupère ces informations et les transmet au Modèle.
    - **Modèle** : Utilise le **pattern Décorateur** pour composer dynamiquement les billets à partir d'un billet de base en ajoutant les caractéristiques souhaitées.
    - **Remarque** : Pour le moment, un client ne peut prendre qu'un billet mais à l'avenir, nous souhaitons qu'ils puissent en prendre plusieurs.
8. **Validation de la réservation**
    - **Vue** : Affiche un récapitulatif de la réservation et propose un bouton de validation.
    - **Contrôleur** : Lors de la validation, demande au Modèle de finaliser et enregistrer la réservation.
    - **Modèle** : Effectue les vérifications finales, enregistre la réservation et met à jour les disponibilités.
    - **Contrôleur** : Informe la Vue du succès (ou de l'échec).
    - **Vue** : Affiche un message de confirmation (ou d'erreur).
9. **Annulation d'une réservation (FUTUR IMPLEMENTATION)**
    - **Vue** : Propose une option « Annuler ma réservation » dans le menu client.
    - **Contrôleur** : Capture la demande d'annulation et transmet l'information (possiblement avec l'identifiant de la réservation) au Modèle.
    - **Modèle** : Vérifie l'existence de la réservation, procède à son annulation et met à jour les disponibilités.
    - **Contrôleur** : Reçoit le résultat et demande à la Vue d'afficher une confirmation d'annulation (ou un message d'erreur si la réservation n'existe pas ou ne peut être annulée).
10. **Nouvelle réservation**
    - **Vue** : Après une réservation validée ou annulée, le menu client reste affiché avec toutes les options.
    - **Contrôleur** : Permet au Client de recommencer une nouvelle réservation en revenant aux étapes 5 à 8 (ou éventuellement via une option dédiée "Nouvelle réservation").
11. **Déconnexion**
    - **Vue** : Propose l'option « Déconnexion » dans le menu client.
    - **Contrôleur** : Intercepte la demande et demande au Modèle de terminer la session utilisateur.
    - **Modèle** : Efface les données de session et réinitialise l'état.
    - **Contrôleur** : Redirige la Vue vers l'écran de connexion ou la page d'accueil, mettant fin à la boucle d'actions.

Scénario détaillé coté Manager

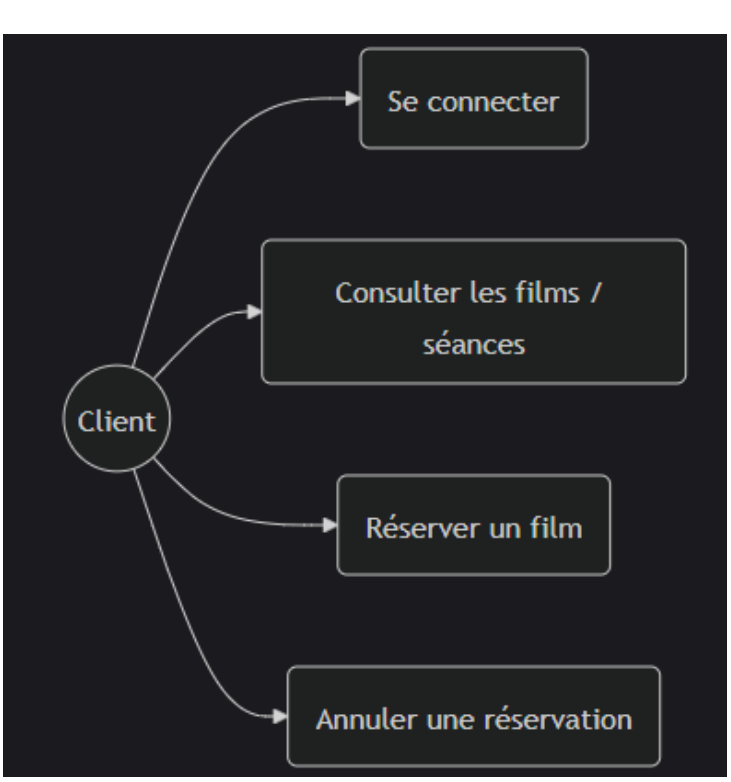
1. **Arrivée au cinéma**
    - **Vue** : Affiche la page d'accueil.
2. **Connexion en tant que Manager**
    - **Vue** : Affiche le formulaire de connexion.
    - **Contrôleur** : Reçoit et transmet les identifiants au Modèle pour vérification.
    - **Modèle** : Vérifie l'authentification et les droits d'accès.
    - **Contrôleur** : Selon le résultat, oriente vers le menu manager ou affiche un message d'erreur.
3. **Affichage du Menu Manager**
    - **Vue** : Affiche un menu dédié au Manager proposant des actions telles que :
        - Créer un film
        - Créer une séance
        - Autres actions de gestion (le cas échéant)
        - Déconnexion
4. **Sélection de l'option "Créer un film"**
    - **Vue** : Le Manager choisit l'option « Créer un film » dans le menu.
    - **Contrôleur** : Capture cette action et affiche un formulaire de création de film.
5. **Saisie et Soumission du Formulaire**
    - **Vue** : Affiche un formulaire permettant de saisir les informations du film (titre, description, durée, etc.).
    - **Utilisateur (Manager)** : Remplit et soumet le formulaire.
6. **Traitement de la Création du Film**
    - **Contrôleur** : Reçoit les données saisies et les transmet au Modèle.
    - **Modèle** : Valide les informations et enregistre le nouveau film dans la base de données.
    - **Contrôleur** : Récupère le résultat du Modèle.
    - **Vue** : Affiche un message de confirmation en cas de succès ou un message d'erreur si la création échoue.
7. **Sélection de l'option "Créer une séance"**
    - **Vue** : Le Manager choisit l'option « Créer une séance » dans le menu.
    - **Contrôleur** : Affiche un formulaire de création de séance.
8. **Saisie et Soumission du Formulaire de Séance**
    - **Vue** : Propose un formulaire pour saisir les détails de la séance (choix du film, horaire, salle, nombre de places, etc.).
    - **Utilisateur (Manager)** : Remplit et soumet le formulaire.
9. **Traitement de la Création de la Séance**
    - **Contrôleur** : Transmet les données saisies au Modèle.
    - **Modèle** : Vérifie la cohérence des informations (par exemple, que le film existe et que l'horaire est disponible) et enregistre la nouvelle séance dans le système.
    - **Contrôleur** : Récupère la réponse du Modèle.
    - **Vue** : Affiche un message de confirmation ou un message d'erreur en fonction du résultat.
10. **Boucle des Actions de Gestion**
    - **Vue** : Après chaque opération (création de film ou de séance), le menu Manager reste affiché.
    - **Contrôleur** : Permet au Manager de réaliser plusieurs actions successives (par exemple, créer plusieurs films ou séances) sans avoir à se reconnecter.
11. **Déconnexion**
    - **Vue** : Propose une option « Déconnexion » dans le menu Manager.
    - **Contrôleur** : Lors de la demande de déconnexion, demande au Modèle de terminer la session.
    - **Modèle** : Efface les données de session et réinitialise l'état.
    - **Contrôleur** : Redirige vers la page de connexion ou l'écran d'accueil.
    - **Vue** : Affiche l'écran d'accueil ou de connexion.

Diagramme de classe : UML

Patron Modèle-Vue-Contrôleur MVC Complet

CF mvc_general.pdf

Diagramme cas d'utilisation



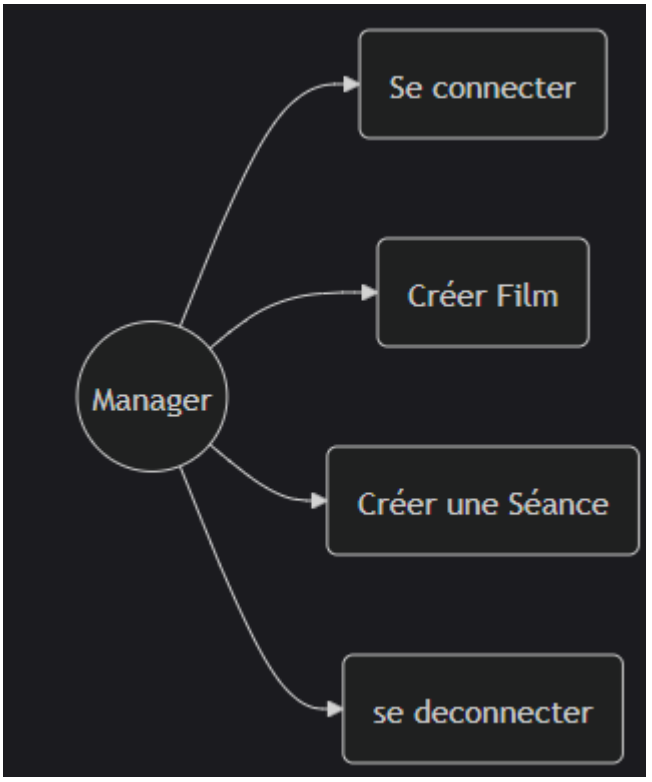
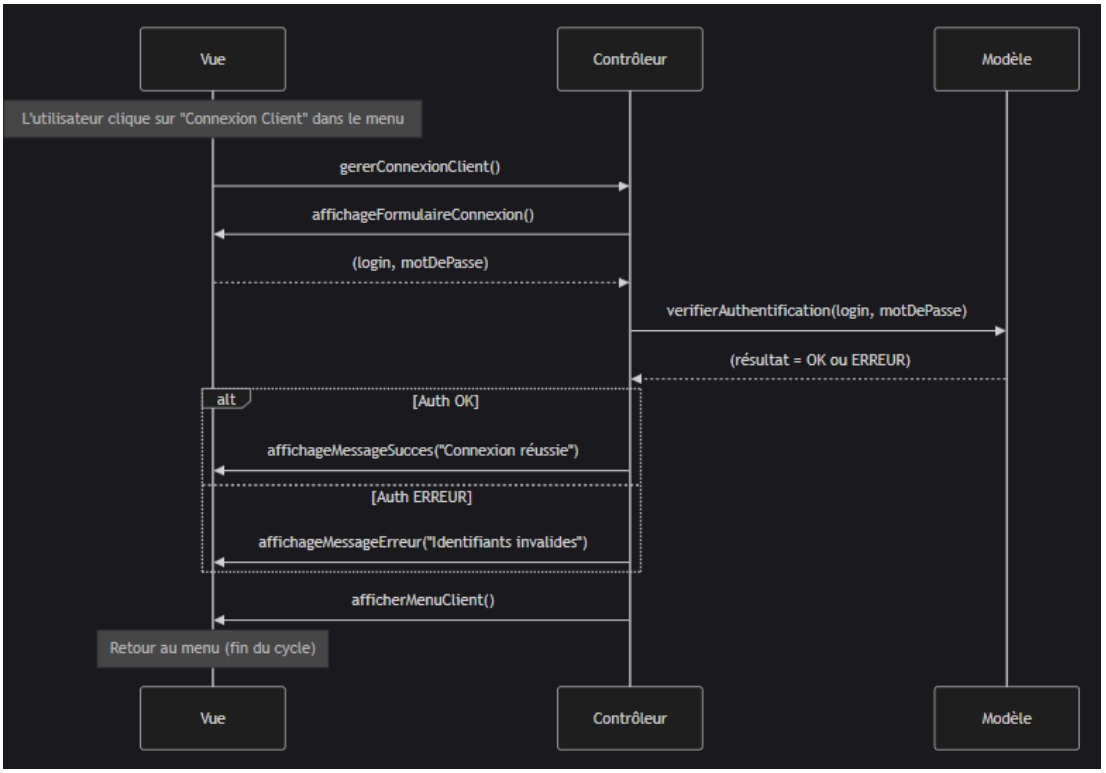
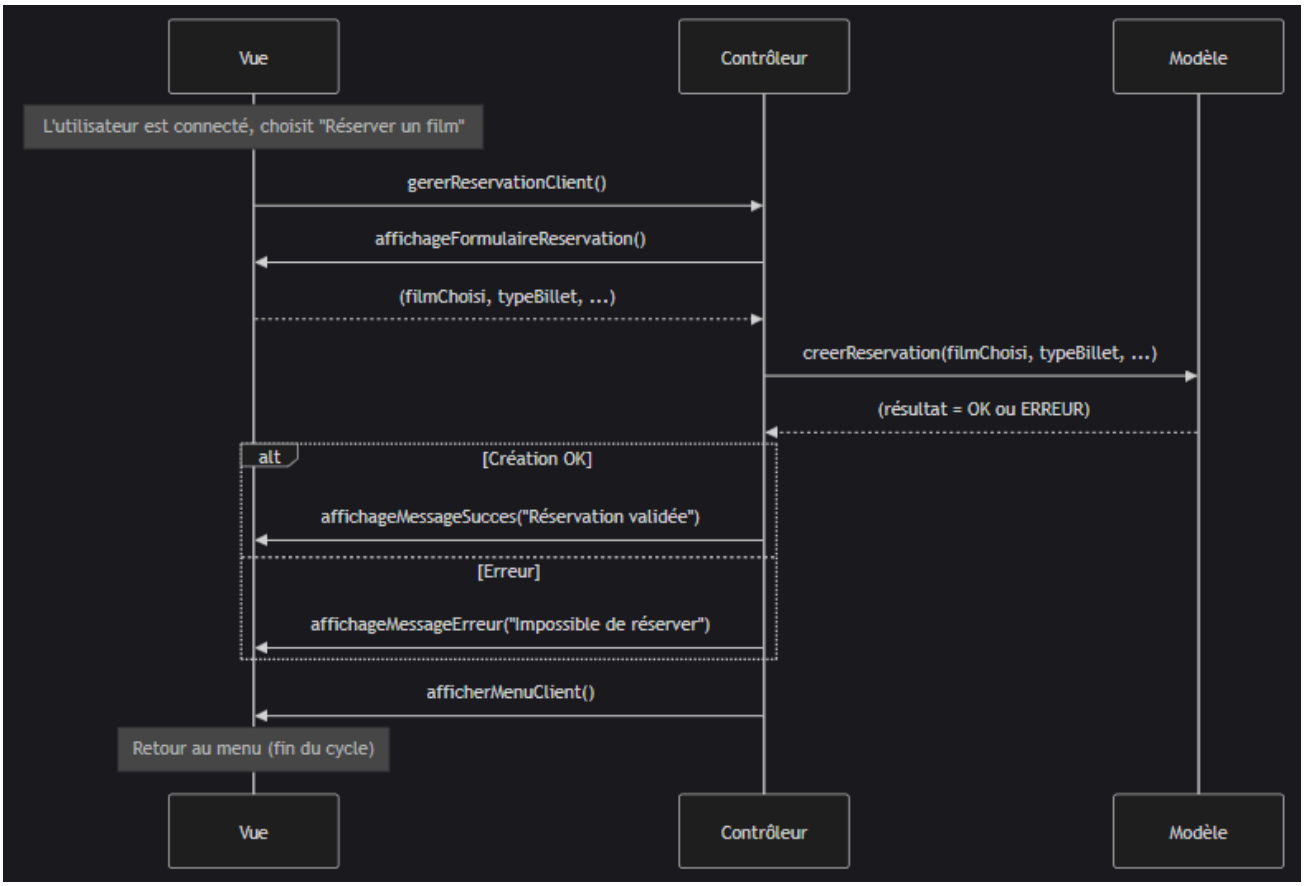


Diagramme de séquences

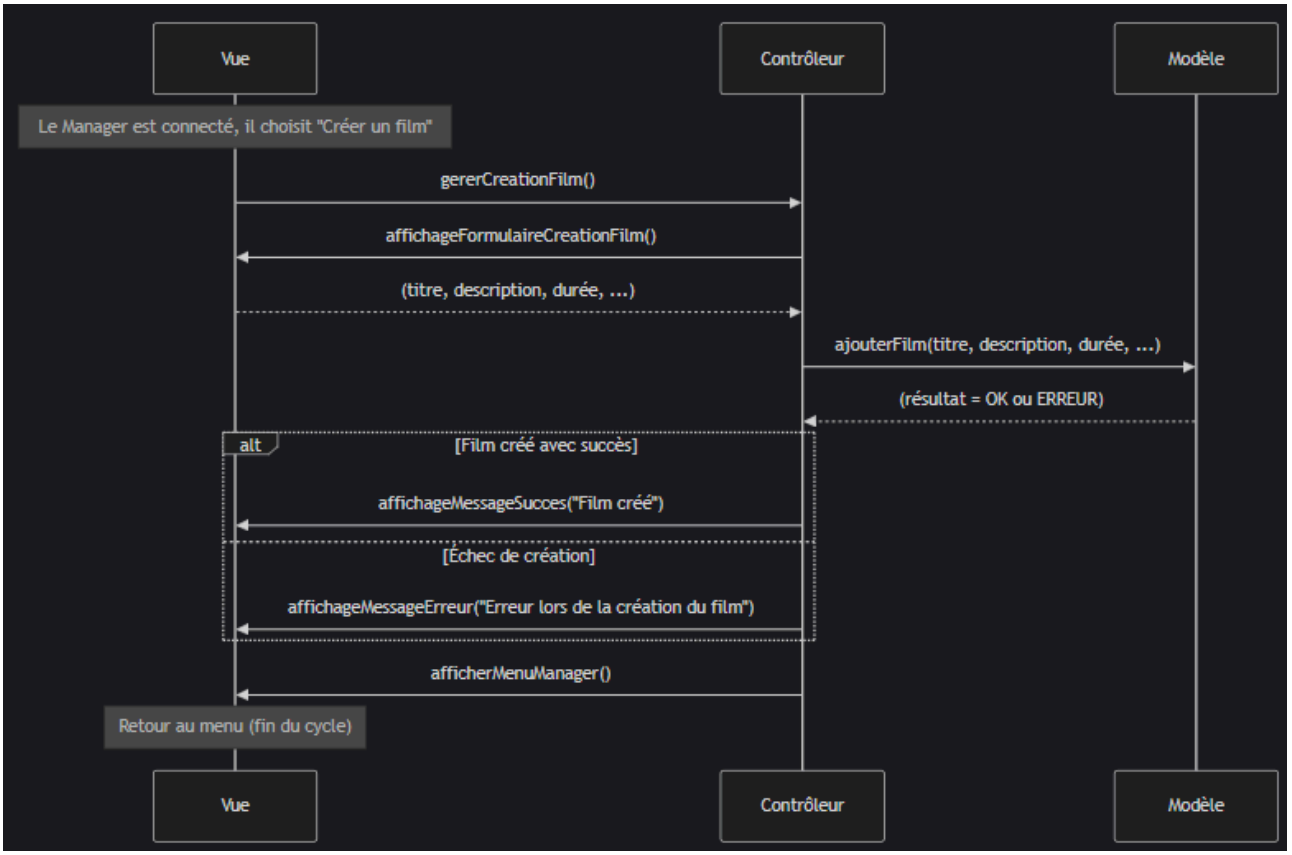
Connexion Client (Déconnexion même schema) :



Réservation d’un film - CLIENT / MANAGER

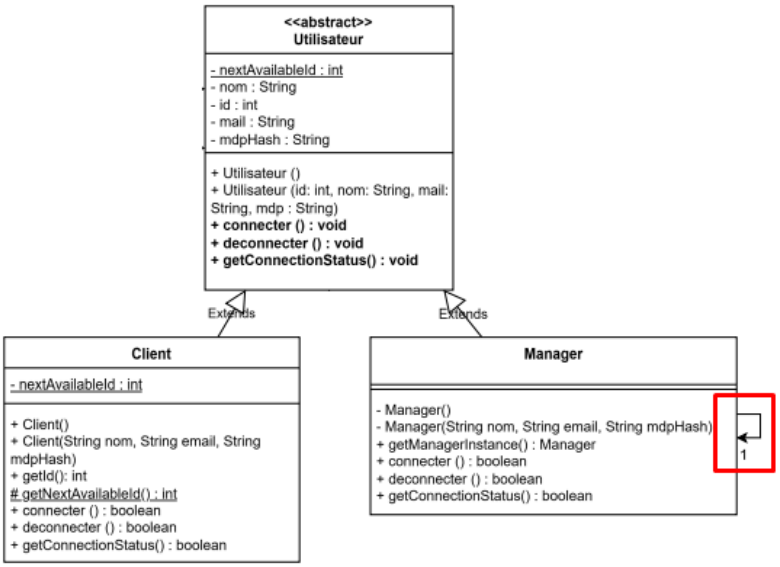


Création d’un Film (Création d’une séance même schéma) - MANAGER



Patrons utilisés et rôles des classes

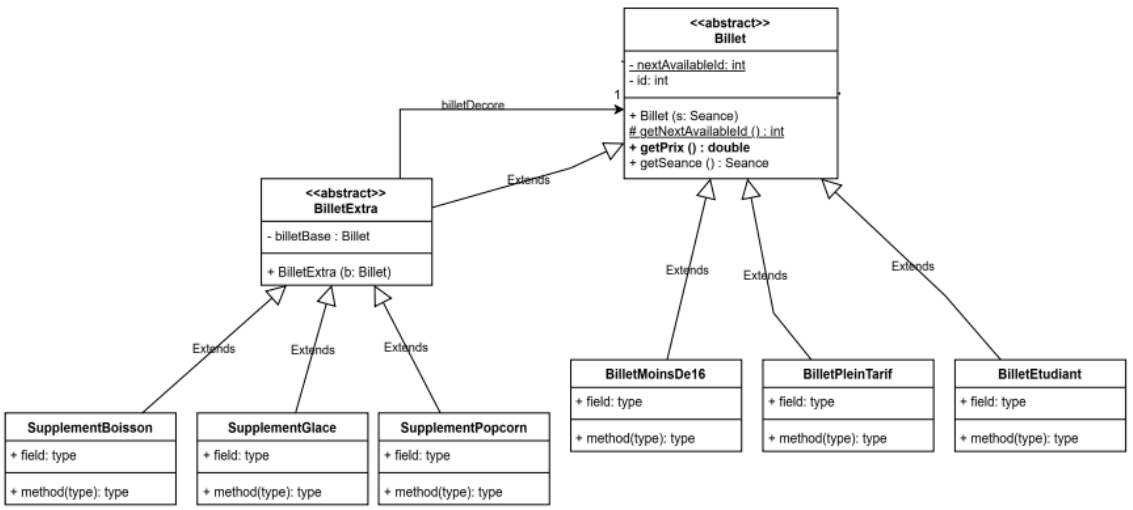
Singleton : Manager



1. Pourquoi un singleton pour Manager ?
L'idée est qu'il n'y ait qu'un seul "manager" pour l'ensemble du système. Le fait de déclarer Manager en singleton garantit que, dans le code, on ne puisse pas instancier plusieurs managers concurrents ou incohérents. On obtient ainsi un point d'accès global (par exemple Manager.getManagerInstance()) pour toutes les opérations qui relèvent de sa responsabilité, et on s'assure de la cohérence de ces opérations (il n'y a qu'un seul "chef d'orchestre" des utilisateurs).

2. Principes SOLID mis en avant dans ce cas
S (Single Responsibility Principle) : chaque classe a une unique responsabilité :
Utilisateur représente un concept générique d'utilisateur (avec un nom, un mail, etc.).
Client gère les spécificités liées aux clients (par exemple la façon dont on les connecte/déconnecte).
Manager gère les spécificités de l'administrateur (singleton, méthodes d'encadrement, etc.).
Ainsi, chaque entité a un rôle clairement séparé et n'a qu'une seule raison de changer.
L (Liskov Substitution Principle) : grâce à l'héritage d'Utilisateur, n'importe quel code qui manipule des objets de type Utilisateur peut accepter indifféremment un Client ou un Manager. On peut substituer une sous-classe à la classe mère sans casser le code existant.
O (Open/Closed Principle) (dans une moindre mesure) : Utilisateur est "ouvert" à l'extension (on peut créer de nouvelles sous-classes), mais "fermé" à la modification (on ne touche pas au code de Utilisateur pour ajouter un nouveau rôle).

Décorateur : Billet



1. Pourquoi utiliser le patron “décorateur” pour Billet ?

Le but est de pouvoir ajouter dynamiquement de nouvelles fonctionnalités ou caractéristiques (des “suppléments”) à un billet de base, sans modifier la classe Billet d’origine.
On a une classe abstraite BilletExtra (le décorateur) qui étend Billet et qui contient un autre Billet (le “billet de base” ou déjà décoré). Chaque “supplément” (SupplementBoisson, SupplementGlace, SupplementPopcorn, etc.) hérite de BilletExtra, et ajoute son propre coût ou son propre comportement en plus de celui du billet “emballé” (wrapped). Ainsi on peut “empiler” plusieurs suppléments : un billet “de base” (moins de 16 ans, plein tarif, étudiant, etc.) auquel on ajoute autant de décorateurs qu’on veut (boisson, pop-corn, glace…).
Ce patron permet une extension flexible : au lieu d’avoir une multitude de sous-classes combinant toutes les variantes (“BilletMoinsDe16EtPopcornEtGlace”, etc.), on a un mécanisme de composition plus modulaire.

2. Principes SOLID mis en avant ?

Single Responsibility Principle (SRP) : Chaque “supplément” est isolé dans une classe dédiée (SupplementBoisson, SupplementGlace, etc.).La classe Billet reste focalisée sur la responsabilité de base : gérer l’identité du billet et sa séance.
Open/Closed Principle (OCP) : La classe Billet est “fermée” à la modification, puisqu’on n’y touche pas pour ajouter des fonctionnalités.Le système est “ouvert” à l’extension : on peut créer de nouveaux BilletExtra pour d’autres suppléments sans changer le code de Billet.
Liskov Substitution Principle (LSP) : Un BilletExtra se substitue à un Billet, puisqu’il hérite de Billet et respecte la même interface (getPrix(), getSeance()).Partout où l’on attend un Billet, on peut passer un décorateur BilletExtra.

MVC : Cinema

1. Pourquoi utiliser le patron MVC pour l’application Cinéma ?

L’objectif est de séparer clairement la logique métier (gestion des films, séances, salles, réservations, utilisateurs, etc.) de la logique d’affichage et de la gestion des interactions. Sur les diagrammes fournis, on observe :
Un Model (via l’interface IModelCinema et son implémentation ModeleCinema) qui encapsule la logique métier : accès aux salles existantes, ajout/suppression de séances, création d’utilisateurs, etc.
Des Views (via les interfaces IVueClient, IVueManager, IVueUtilisateur et leurs implémentations VueClient, VueManager, VueCinema, etc.) chargées de présenter l’information (affichage des films, formulaire de réservation, etc.) et de collecter les saisies de l’utilisateur (boutons cliqués, champs remplis, etc.).
Un Controller (via l’interface IControleurCinema et son implémentation ControleurCinema) qui fait le lien entre la View et le Model : il réceptionne les actions de l’utilisateur (demande de connexion, création d’une séance, réservation d’un billet…), interroge/met à jour le Model, et actualise la View en conséquence.

NB : Grâce à ce découpage, chacune de ces couches peut évoluer ou être remplacée sans impact majeur sur les autres : On peut ajouter une nouvelle IHM (par exemple une interface web ou une appli mobile) sans toucher à la logique métier dans ModeleCinema. On peut faire évoluer les règles métier (par exemple la manière de calculer un tarif) en gardant la même interface graphique. Le contrôleur orchestre simplement les échanges, évitant que la View “sache” trop de choses sur le Model ou inversement.

2. Principes SOLID mis en avant ?

Single Responsibility Principle (SRP) : Le Model (ModeleCinema) se concentre sur la gestion des entités du domaine (films, salles, réservations…). Les Views (VueCinema, VueClient, VueManager…) se consacrent à l’affichage et à l’interaction avec l’utilisateur. Le Controller (ControleurCinema) orchestre l’ensemble. Ainsi, chaque couche a sa responsabilité unique.
Open/Closed Principle (OCP) : Les interfaces (IModelCinema, IVueClient, IVueManager, IControleurCinema) permettent d’ouvrir le système à de nouvelles implémentations (on peut créer d’autres vues, d’autres modèles) sans modifier les classes existantes.
Liskov Substitution Principle (LSP) : Les contrôleurs et vues reposent sur des interfaces. Tant qu’une classe implémente correctement l’interface, elle peut être substituée sans casser le fonctionnement (par ex. n’importe quelle classe qui implémente IVueManager peut remplacer VueManager).
Interface Segregation Principle (ISP) : Au lieu d’avoir une unique interface énorme pour la vue, on dispose de plusieurs interfaces spécialisées (IVueClient, IVueManager, IVueUtilisateur). Les composants n’ont donc accès qu’aux méthodes dont ils ont besoin.
Dependency Inversion Principle (DIP) : Le contrôleur dépend d’abstractions (IModelCinema, IVueClient, IVueManager) et non d’implémentations concrètes. Cela rend le code plus flexible et testable (on peut injecter des “mock” ou “stub” pour les tests).

Implémentations futures et ajouts potentielles

Évolutions autour du Décorateur “Billet”

- On peut ajouter autant de nouveaux Supplement... que souhaité : par exemple, “3D”, “Fauteuil VIP”, “Accès exclusif”, “Réduction spéciale abonné”, etc.
- Chaque supplément reste une sous-classe de BilletExtra ; pas besoin de toucher à la classe de base Billet. Cela illustre bien le principe d’Ouverture/Fermeture (OCP).

Évolutions autour de la class ModeleCinema en singleton

Instance unique : En restreignant ModeleCinema à un seul objet dans l’application, on évite qu’une même réservation soit gérée par plusieurs instances différentes, ce qui pourrait créer des conflits ou des incohérences.
Point d’accès global : Toutes les opérations de réservation s’effectuent à travers cette unique instance. Les modifications sont ainsi centralisées et cohérentes.
Mise en œuvre : Le constructeur de ModeleCinema devient privé et la classe fournit une méthode statique getInstance() qui crée l’instance une seule fois, puis retourne toujours la même référence.