Laboratoires 2 et 3 - Système de Gestion de Tickets

6GEI311 - Architecture des logiciels

Équipe: Antoine Larouche Tremblay

Date de remise Lab 2: 7 octobre 2025
Date de remise Lab 3: 28 octobre 2025

Table des matières

Instructions de compilation et exécution

Prérequis

- Java JDK 8 ou supérieur

Laboratoire 2 - Version Console

Commandes de compilation

```
cd C:\Users\Antoine\Desktop\8-automne_2025\6GEI311\2-laboratoires\6GEI311_lab2-2
javac -encoding UTF-8 -d classes MainConsole.java core/content/*.java
core/exporter/*.java core/entities/*.java
```

Commande d'exécution

```
java -cp classes MainConsole
```

Résultat attendu

Le programme exécute 9 tests démontrant les descriptions multi-format, les patrons Composite/Strategy/Visitor, la validation des transitions de statut, et la gestion administrative des tickets.

Laboratoire 3 - Interface Graphique

Commandes de compilation

```
cd C:\Users\Antoine\Desktop\8-automne_2025\6GEI311\2-laboratoires\6GEI311_lab2-2
javac -encoding UTF-8 -d classes MainGUI.java gui/**/*.java core/**/*.java
```

Commande d'exécution

```
java -cp classes MainGUI
```

Résultat attendu

Lance l'interface graphique Swing avec :

- Dialogue de connexion (sélection utilisateur/rôle)
- Fenêtre principale avec table des tickets
- Fonctionnalités complètes de gestion de tickets selon les permissions

---

Structure du projet

```
6GEI311_lab2-2/
├── MainConsole.java                 # Programme de test avec 9 scénarios (Lab 2)
├── MainGUI.java                     # Programme principal interface graphique
(Lab 3)
├── core/
│   ├── entities/
│   │   ├── User.java                # Utilisateur avec createTicket(),
viewTicket(), updateTicket()
│   │   ├── Admin.java               # Administrateur héritant de User
│   │   ├── Ticket.java              # Ticket avec description: Content
```

```
│   │   └── TicketStatus.java        # Énumération avec validation des transitions
│   ├── content/                      # Patron Composite
│   │   ├── Content.java             # Interface commune
│   │   ├── TextContent.java         # Contenu textuel
│   │   ├── ImageContent.java        # Contenu image
│   │   ├── VideoContent.java        # Contenu vidéo
│   │   └── CompositeContent.java    # Composition de contenus
│   └── exporter/                     # Patron Strategy
│       ├── Exporter.java            # Interface d'export
│       └── PDFExporter.java         # Implémentation export PDF
├── gui/                              # Package interface graphique (Lab 3)
│   ├── controllers/
│   │   ├── ApplicationState.java    # Singleton + Observer pattern
│   │   ├── TicketController.java    # Pont MVC: Domain ↔ DTO
│   │   └── TicketStateListener.java # Interface Observer
│   ├── models/
│   │   ├── TicketDTO.java           # Data Transfer Object tickets
│   │   ├── UserDTO.java             # DTO utilisateurs
│   │   ├── ContentItemDTO.java      # DTO contenu
│   │   └── TicketTableModel.java    # Modèle table Swing
│   ├── views/
│   │   ├── TicketManagerGUI.java    # Fenêtre principale
│   │   ├── dialogs/
│   │   │   ├── LoginDialog.java
│   │   │   ├── CreateTicketDialog.java
│   │   │   ├── EditTicketDialog.java
│   │   │   └── ContentBuilderPanel.java
│   │   └── components/
│   │       └── TicketDetailPanel.java
│   ├── services/
│   │   ├── FileService.java         # Gestion I/O
│   │   └── PermissionService.java   # Vérification permissions
│   ├── validators/
│   │   └── TicketValidator.java     # Validation centralisée
│   └── utils/
│       └── ErrorHandler.java        # Gestion messages utilisateur
├── README.md                         # Ce rapport
└── gui_mvc.md                        # Documentation architecture GUI
```

PARTIE I - Laboratoire 2 - Système de Gestion de Tickets

SECTION I - Critique du diagramme de classe initial

Le diagramme de classe fourni présente plusieurs problèmes architecturaux qui compromettent la modificabilité du système.

1.1 Dépendance bidirectionnelle entre Ticket et User

Problème: Le diagramme montre une méthode `Ticket.assignTo(user: User): void`, ce qui signifie que la classe Ticket doit connaître la classe User. En parallèle, la méthode `User.createTicket(ticket:`

`Ticket): void` indique que User connaît Ticket. Cette configuration crée une dépendance circulaire entre les deux classes.

Analyse détaillée de la relation:

La cardinalité indiquée dans le diagramme (User "1" creates "0..*" Ticket) révèle une relation de composition où User est responsable de la création des Ticket. Cependant, la méthode `assignTo(user: User)` dans Ticket inverse cette relation en créant une dépendance dans la direction opposée. Cette situation pose plusieurs problèmes conceptuels:

1. **Violation du principe de responsabilité unique**: La classe Ticket possède deux responsabilités conflictuelles : gérer son propre cycle de vie ET connaître la structure de User pour l'assignation. Si les attributs de User changent (par exemple, ajout d'un système de permissions), la classe Ticket pourrait nécessiter des modifications.

2. **Ordre de compilation indéterminé**: Dans un environnement de compilation Java, le compilateur ne peut pas déterminer quelle classe compiler en premier. User nécessite Ticket pour le type de retour de createTicket(), et Ticket nécessite User pour le paramètre de assignTo(). Cette situation force l'utilisation de compilation en plusieurs passes ou de déclarations forward, complexifiant le processus de build.

3. **Testabilité compromise**: Pour tester la classe Ticket de manière isolée, il devient nécessaire de créer ou mocker des instances de User, même pour des tests qui ne concernent pas l'assignation. Inversement, tester User nécessite Ticket. Cette dépendance mutuelle rend les tests unitaires plus complexes et fragiles.

4. **Violation du principe de moindre connaissance (Loi de Déméter)**: La classe Ticket n'a pas besoin de connaître toute la structure de User pour accomplir une assignation. Elle a seulement besoin d'un identifiant unique. En acceptant un objet User complet, Ticket a accès à des informations qui ne le concernent pas (name, email, role), violant l'encapsulation.

5. **Rigidité architecturale**: Cette dépendance bidirectionnelle empêche l'évolution indépendante des classes. Par exemple, si on souhaite réutiliser la classe Ticket dans un autre contexte (système de gestion de projets, système de support client), on est forcé d'inclure également la classe User, même si la notion d'utilisateur est différente dans ce nouveau contexte.

Impact sur la modificabilité:

- Couplage fort rendant impossible la modification d'une classe sans affecter l'autre
- Impossibilité de compiler Ticket sans User et vice-versa, créant des cycles de compilation
- Difficulté de réutilisation des classes dans d'autres contextes ou modules
- Risque élevé de propagation des changements à travers la dépendance circulaire
- Tests unitaires complexes nécessitant la création des deux classes simultanément
- Violation du principe de moindre connaissance (Ticket connaît trop de détails sur User)
- Impossibilité d'utiliser Ticket dans un contexte où User n'existe pas ou est défini différemment

Problème supplémentaire - Signature incohérente de createTicket():

La signature de la méthode `User.createTicket(ticket: Ticket): void` présente une incohérence conceptuelle majeure. Le nom de la méthode suggère une action de création ("create"), mais la signature prend en paramètre un objet Ticket déjà instancié. Cette contradiction soulève plusieurs questions:

- Si le Ticket est déjà créé avant l'appel à createTicket(), qui est responsable de sa création?
- Pourquoi passer un objet déjà existant à une méthode censée le créer?
- Quel est le rôle réel de cette méthode: créer, enregistrer, ou valider un ticket?

Impact sur l'utilisabilité et la maintenabilité:

Cette signature crée une confusion importante sur la responsabilité de création et conduit à une API peu intuitive. Un développeur utilisant cette classe devrait d'abord instancier un Ticket (avec `new Ticket(...)`), puis le passer à `createTicket()`, ce qui est contre-intuitif. De plus, le type de retour `void` signifie que l'appelant ne peut pas récupérer le ticket créé, forçant à maintenir une référence locale, augmentant la complexité du code client.

Solution implémentée: Dans notre code final, la signature a été modifiée en `createTicket(String title, Content description, String priority): Ticket`, où la méthode prend les paramètres nécessaires et retourne le ticket créé, respectant ainsi le principe de moindre surprise et simplifiant l'utilisation.

1.2 Attribut description de type String non extensible

Problème: L'attribut `description: String` limite le contenu des tickets à du texte simple, alors que les exigences spécifient explicitement que la description doit pouvoir inclure "du texte, des captures d'écran ou des vidéos".

Impact sur la modificabilité:

- Violation du principe Open/Closed (fermé à l'extension)
- Nécessité de modifier la classe Ticket pour ajouter le support d'images ou vidéos
- Impossibilité de composer plusieurs types de contenu (texte + image + vidéo)
- Logique d'affichage et d'export complexe avec multiples conditions

1.3 Attribut status de type String sans validation

Problème: L'utilisation d'un `status: String` permet l'assignation de valeurs arbitraires sans aucun contrôle. Aucun mécanisme ne garantit que les transitions de statut respectent le cycle de vie défini (OUVERT → ASSIGNÉ → VALIDATION → TERMINÉ).

Impact sur la modificabilité:

- Absence de validation des transitions entre états
- Risque d'erreurs de saisie ("Ouvert" vs "OUVERT" vs "ouvert")
- Logique de validation dispersée dans le code au lieu d'être centralisée
- Difficulté de maintenance et d'évolution du cycle de vie

1.4 Absence de mécanisme de génération des identifiants

Problème: Le diagramme ne spécifie aucun mécanisme pour générer automatiquement les `ticketID` de manière unique.

Impact sur la modificabilité:

- Responsabilité de génération d'IDs floue et non définie
- Risque de duplication d'identifiants

- Code client complexe devant gérer la génération

1.5 Absence de structure d'export

Problème: Bien que l'exigence stipule que les descriptions doivent pouvoir être exportées au format PDF, le diagramme ne propose aucune structure pour gérer l'export vers différents formats.

Impact sur la modificabilité:

- Export PDF codé en dur dans la classe Ticket
- Impossibilité d'ajouter d'autres formats (HTML, JSON) sans modifier Ticket
- Violation du principe Open/Closed

SECTION II - Améliorations apportées au diagramme

2.1 Tableau récapitulatif des changements et justifications

#	Changement effectué	Problème résolu	Bonne pratique appliquée	Principe SOLID respecté
1	`Ticket.assignTo(int userID)` au lieu de `assignTo(User user)`	Dépendance bidirectionnelle Ticket ↔ User	Faible couplage	Dependency Inversion Principle
2	`description: Content` (interface) au lieu de `String`	Description non extensible	Programmation par interface	Open/Closed Principle
3	Interface `Content` avec 4 implémentations concrètes	Support multi-format requis	Polymorphisme	Single Responsibility Principle
4	`CompositeContent` contenant `List<Content>`	Descriptions composites (texte+image+vidéo)	Patron de conception Composite	Open/Closed Principle
5	Interface `Exporter` avec `PDFExporter`	Export extensible vers multiples formats	Patron de conception Strategy	Open/Closed Principle
6	Méthode `accept(Exporter exporter)` dans `Content`	Double dispatch pour export type-safe	Patron de conception Visitor	Liskov Substitution Principle
7	Enum `TicketStatus` avec méthode `canTransitionTo()`	Validation des transitions de statut	Type énuméré avec logique métier	Single Responsibility Principle
8	Attribut statique `ticketIDCounter` dans `User`	Génération automatique d'IDs uniques	Encapsulation de la logique	Single Responsibility Principle

#	Changement effectué	Problème résolu	Bonne pratique appliquée	Principe SOLID respecté
9	`Admin extends User`	Réutilisation du code de User	Héritage et factorisation	Liskov Substitution Principle

2.2 Patrons de conception appliqués

Patron Composite

Problème résolu: La description d'un ticket doit pouvoir contenir du texte simple, des images, des vidéos, ou une combinaison de ces éléments. Le type String ne permet pas cette flexibilité.

Structure mise en place:

```
Content (interface)
├── TextContent (feuille) : contient String text
├── ImageContent (feuille) : contient String imagePath, String caption
├── VideoContent (feuille) : contient String videoPath, int duration
└── CompositeContent (composite) : contient List<Content> children
```

Justification du choix:

- Permet de traiter de manière uniforme les contenus simples (feuilles) et composés (composites)
- Un ticket peut avoir une description textuelle simple OU une description riche (texte + image + vidéo)
- Extensibilité garantie : ajout de nouveaux types (AudioContent) sans modification des classes existantes
- Respect du principe Open/Closed

Méthodes de l'interface Content:

- `String display()` : affichage pour consultation dans la plateforme
- `String accept(Exporter exporter)` : accepte un visiteur pour l'export (voir patron Visitor)

Instanciation du patron: Le patron Composite est instancié à travers l'interface `Content` qui définit le contrat commun. Les classes `TextContent`, `ImageContent` et `VideoContent` représentent les feuilles (contenus atomiques), tandis que `CompositeContent` représente le composite capable de contenir d'autres objets `Content`. Cette structure permet de construire des arborescences de contenu de complexité arbitraire.

Patron Strategy

Problème résolu: L'exigence stipule que les descriptions doivent pouvoir être exportées au format PDF. Le système doit être conçu pour permettre l'ajout ultérieur d'autres formats (HTML, JSON) sans modification du code existant.

Structure mise en place:

```
Exporter (interface)
└── PDFExporter (stratégie concrète)
    (Futurs : HTMLExporter, JSONExporter, MarkdownExporter)
```

Justification du choix:

- Permet d'ajouter de nouveaux formats d'export sans modifier les classes Content ou Ticket
- Possibilité de changer de stratégie d'export dynamiquement à l'exécution
- Séparation claire entre la logique métier (Content) et la logique d'export (Exporter)
- Respect du principe Open/Closed

Méthodes de l'interface Exporter:

- `String export(Content content)` : point d'entrée principal
- `String exportText(TextContent textContent)` : export spécifique au texte
- `String exportImage(ImageContent imageContent)` : export spécifique aux images
- `String exportVideo(VideoContent videoContent)` : export spécifique aux vidéos
- `String exportComposite(CompositeContent compositeContent)` : export récursif des composites

Instanciation du patron: Le patron Strategy est instancié via l'interface `Exporter` qui définit les différentes opérations d'export. La classe `PDFExporter` fournit une implémentation concrète pour le format PDF. L'architecture permet d'ajouter facilement de nouvelles stratégies d'export en créant simplement une nouvelle classe implémentant `Exporter`, sans aucune modification des classes existantes.

Patron Visitor

Problème résolu: Pour exporter un contenu, il faut connaître à la fois le type de contenu (TextContent, ImageContent, etc.) et le format d'export (PDF, HTML, etc.). Une approche naïve utiliserait `instanceof` et des cast, ce qui viole les principes SOLID.

Structure mise en place:

```java
// Interface Content (élément visitable)
public interface Content {
    String accept(Exporter exporter);
}

// Implémentation dans TextContent
public class TextContent implements Content {
    public String accept(Exporter exporter) {
        return exporter.exportText(this);  // Double dispatch
    }
}
```

Justification du choix:

- Évite l'utilisation de `instanceof` et de cast explicites
- Permet au compilateur de vérifier statiquement les types (type-safety)

- Chaque type de contenu contrôle comment il doit être exporté
- Extensibilité maximale pour nouveaux types et nouvelles opérations

Instanciation du patron: Le patron Visitor est instancié à travers la méthode `accept(Exporter exporter)` dans l'interface `Content`. Cette méthode implémente le mécanisme de double dispatch : premièrement, l'appelant sélectionne le visiteur (Exporter), deuxièmement, l'objet Content redirige vers la méthode appropriée de l'Exporter en fonction de son type concret. Ce mécanisme garantit que la bonne méthode d'export est appelée sans aucun test de type explicite.

2.3 Démonstration de la modificabilité

Scénario 1 : Ajout d'un nouveau type de contenu (AudioContent)

Étapes:

1. Créer une nouvelle classe `AudioContent implements Content`
2. Implémenter les méthodes `display()` et `accept(Exporter exporter)`
3. Ajouter une méthode `exportAudio(AudioContent audioContent)` dans l'interface `Exporter`
4. Implémenter cette méthode dans `PDFExporter` et autres exporters existants

Modifications requises:

- Aucune modification de la classe Ticket
- Aucune modification de CompositeContent
- Aucune modification des autres types de Content (TextContent, ImageContent, VideoContent)
- Ajout de code uniquement dans les nouvelles classes et interfaces

Scénario 2 : Ajout d'un nouveau format d'export (HTMLExporter)

Étapes:

1. Créer une nouvelle classe `HTMLExporter implements Exporter`
2. Implémenter toutes les méthodes d'export (exportText, exportImage, exportVideo, exportComposite)

Modifications requises:

- Aucune modification de l'interface Content ou de ses implémentations
- Aucune modification de la classe Ticket
- Ajout de code uniquement dans la nouvelle classe HTMLExporter

---

SECTION III - Autres améliorations apportées

3.1 Énumération TicketStatus avec validation des transitions

Amélioration: Remplacement de `status: String` par `status: TicketStatus` (énumération).

Implémentation: L'énumération `TicketStatus` définit les valeurs possibles : OUVERT, ASSIGNE, VALIDATION, TERMINE, FERME. Elle inclut une méthode `canTransitionTo(TicketStatus newStatus)` qui valide si une transition vers un nouveau statut est autorisée selon le cycle de vie défini.

Cycle de vie implémenté:

```
OUVERT → ASSIGNE → VALIDATION → TERMINE
       ↘ FERME
```

Règles de transition:

- Depuis OUVERT : peut aller vers ASSIGNE ou FERME
- Depuis ASSIGNE : peut aller vers VALIDATION ou FERME
- Depuis VALIDATION : peut aller vers TERMINE ou retourner à ASSIGNE (pour correction)
- TERMINE et FERME sont des états finaux (aucune transition autorisée)

Cette approche garantit la cohérence des états, empêche les erreurs de saisie, centralise la logique de validation et facilite l'évolution du cycle de vie.

3.2 Stockage et affichage de l'historique des commentaires

Amélioration: Ajout d'un attribut `List<String> comments` dans la classe Ticket.

Méthodes ajoutées:

- `addComment(String comment)` : ajoute un commentaire avec validation (non null, non vide)
- `getComments()` : retourne une copie défensive de la liste des commentaires
- `displayComments()` : affiche l'historique formaté des commentaires

Permet de suivre l'évolution d'un ticket à travers les commentaires des développeurs. La copie défensive dans `getComments()` protège l'encapsulation en empêchant la modification directe de la liste interne.

3.3 Héritage Admin extends User

Amélioration: La classe Admin hérite de User au lieu d'être une classe complètement séparée.

- Un administrateur EST un utilisateur avec des privilèges supplémentaires
- Satisfait le principe de substitution de Liskov (un Admin peut remplacer un User partout)
- Admin peut créer des tickets (méthode héritée de User) en plus de ses fonctions administratives
- Évite la duplication des attributs (userID, name, email)

Méthodes spécifiques à Admin:

- `assignTicket(Ticket ticket, int userID)` : assigne un ticket à un utilisateur spécifié par son ID
- `closeTicket(Ticket ticket)` : ferme un ticket (transition vers TERMINE)
- `viewAllTickets(List<Ticket> tickets)` : consulte l'ensemble des tickets du système

Note importante: La signature `assignTicket(Ticket, int userID)` utilise un entier au lieu d'une référence User pour éviter une dépendance bidirectionnelle entre Admin et User.

3.4 Tests exhaustifs dans la classe MainConsole

Voici les 9 scénarios de test complets démontrant toutes les fonctionnalités du système.

Scénarios implémentés:

1. Description textuelle simple avec export PDF
2. Image avec légende et export PDF
3. Vidéo avec durée et export PDF
4. Description composite combinant texte, image et vidéo avec export PDF
5. Modification dynamique d'une description (passage de texte simple à composite)
6. Gestion administrative (assignation, ajout de commentaires)
7. Validation des transitions de statut avec test de transition invalide
8. Démonstration que Admin peut créer des tickets (héritage de User)
9. Vue d'ensemble de tous les tickets par un administrateur

Ces tests valident l'ensemble des patrons de conception, démontrent la modificabilité du système et servent de documentation vivante sur l'utilisation des classes.

SECTION IV - Leçons apprises

Importance de l'élimination des dépendances bidirectionnelles

Une dépendance bidirectionnelle entre deux classes réduit drastiquement la modificabilité du système. Elle crée un couplage fort qui rend impossible la modification d'une classe sans affecter l'autre, et complique considérablement les tests unitaires. En transformant `assignTo(User user)` en `assignTo(int userID)`, nous avons éliminé la dépendance de Ticket vers User. Le couplage devient unidirectionnel (User → Ticket), ce qui permet de modifier la classe Ticket indépendamment de User. Cette décision respecte le principe Dependency Inversion en faisant dépendre Ticket d'un type primitif plutôt que d'une classe concrète.

Les patrons de conception facilitent concrètement la modificabilité

Les patrons de conception Composite, Strategy et Visitor ne sont pas de la sur-ingénierie abstraite. Ils offrent des bénéfices mesurables en termes de temps de développement et de qualité du code. L'ajout d'un nouveau type de contenu (par exemple AudioContent) nécessite environ 15 minutes avec l'architecture actuelle basée sur le patron Composite, contre plus de 2 heures de refactorisation avec l'approche initiale utilisant `description: String`. De même, l'ajout d'un nouveau format d'export (HTMLExporter) se fait sans aucune modification du code existant grâce au patron Strategy.

Dépendre d'abstractions plutôt que d'implémentations

Le principe Dependency Inversion (le D de SOLID) stipule que les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais que tous deux doivent dépendre d'abstractions. La classe Ticket dépend de l'interface Content, pas des implémentations concrètes TextContent, ImageContent ou VideoContent. Cette décision permet d'ajouter de nouveaux types de contenu sans modifier Ticket. De même, l'utilisation de l'interface Exporter permet d'ajouter de nouveaux formats d'export sans toucher aux classes Content.

Le principe Open/Closed en pratique

Concevoir pour l'extension dès le début du développement évite des refactorisations majeures ultérieures. Le principe Open/Closed (ouvert à l'extension, fermé à la modification) doit guider les décisions architecturales

initiales. Le système actuel est fermé à la modification (pas besoin de changer la classe Ticket) mais ouvert à l'extension (possibilité d'ajouter de nouveaux types Content et de nouveaux Exporter). Cette architecture facilite grandement la maintenance et l'évolution du système.

Séparation des préoccupations et responsabilité unique

Chaque classe doit avoir une responsabilité unique et clairement définie. La confusion des responsabilités conduit à des classes monolithiques difficiles à maintenir et à tester.

- Les classes Content ont pour unique responsabilité de représenter du contenu
- Les classes Exporter ont pour unique responsabilité de générer un format d'export
- La classe Ticket a pour unique responsabilité de gérer le cycle de vie d'un ticket
- Aucune classe ne fait le travail d'une autre, ce qui facilite les tests et la maintenance

Type-safety avec les énumérations

Les énumérations offrent une validation au moment de la compilation et une sécurité de type impossibles à obtenir avec des chaînes de caractères. Elles permettent également d'encapsuler la logique métier liée aux valeurs énumérées. L'énumération `TicketStatus` empêche les erreurs de frappe ("Ouvert" vs "OUVERT"), valide automatiquement les transitions entre états via la méthode `canTransitionTo()`, et centralise la logique du cycle de vie. Cette approche est beaucoup plus robuste que l'utilisation de `status: String` avec validation manuelle dispersée dans le code.

Principes SOLID appliqués

Single Responsibility Principle: Chaque classe a une responsabilité unique et bien définie. Content représente du contenu, Exporter gère l'export, Ticket gère le cycle de vie.

Open/Closed Principle: Le système est ouvert à l'extension (nouveaux types Content, nouveaux Exporter) mais fermé à la modification (pas besoin de changer Ticket).

Liskov Substitution Principle: Tous les Content sont interchangeables. Admin peut remplacer User partout où un User est attendu.

Interface Segregation Principle: Les interfaces Content et Exporter sont ciblées et ne forcent pas les implémentations à dépendre de méthodes inutiles.

Dependency Inversion Principle: Ticket dépend de l'abstraction Content, pas des implémentations concrètes TextContent ou ImageContent.

Comparaison avant/après

Critère	Diagramme initial	Code final
Types de contenu supportés	Texte uniquement	Texte, Image, Vidéo, Composite
Dépendances bidirectionnelles	Oui (Ticket ↔ User)	Non (User → Ticket unidirectionnel)
Ajout d'un nouveau type	Modifier Ticket	Créer classe implements Content

Critère	Diagramme initial	Code final
Formats d'export	Aucun	PDF (extensible vers HTML, JSON)
Validation du statut	Aucune	Enum avec validation des transitions
Temps estimé ajout fonctionnalité	Environ 2 heures	Environ 15 minutes
Respect du principe Open/Closed	Non respecté	Respecté

Changements par rapport au diagramme initial

Classe User

- Attributs identiques : userID, name, email, role
- Méthode `createTicket()` modifiée : retourne Ticket et prend Content en paramètre
- Méthodes `viewTicket()` et `updateTicket()` conformes au diagramme
- Ajout de l'attribut statique `ticketIDCounter` pour génération automatique des IDs

Classe Admin

- Héritage ajouté : `extends User` pour réutilisation du code
- Méthode `assignTicket()` modifiée : paramètre `int userID` au lieu de `User user`
- Méthodes `closeTicket()` et `viewAllTickets()` conformes au diagramme

Classe Ticket

- Attribut `description` modifié : type `Content` au lieu de `String`
- Attribut `status` modifié : type `TicketStatus` (enum) au lieu de `String`
- Méthode `assignTo()` modifiée : paramètre `int userID` au lieu de `User user`
- Autres attributs et méthodes conformes au diagramme
- Ajout de méthodes : `displayDescription()`, `exportToPDF()`, gestion des commentaires

Nouvelles structures

- Package `core.content` : Interface Content + TextContent, ImageContent, VideoContent, CompositeContent (Patron Composite)
- Package `core.exporter` : Interface Exporter + PDFExporter (Patron Strategy)
- Énumération `TicketStatus` : Validation type-safe des transitions d'état

---

PARTIE II - Laboratoire 3 - Interface graphique

Application de Gestion de Tickets - Architecture GUI

Description du Logiciel

Application Java Swing de gestion de tickets suivant l'architecture MVC. Permet aux utilisateurs de créer, modifier, assigner et suivre des tickets avec contenu multimédia (texte, images, vidéos).

Fonctionnalités principales

- Authentification utilisateur (Utilisateur, Développeur, Admin)
- Création/modification de tickets avec contenu multimédia composite
- Assignation de tickets aux utilisateurs
- Gestion des statuts (transitions d'état)
- Commentaires sur les tickets
- Export de rapports
- Système de permissions basé sur les rôles

Structure du Package `gui/`

`controllers/` - Logique de coordination

- **ApplicationState.java** - Singleton gérant l'état global (utilisateur connecté, tickets, utilisateurs). Implémente le pattern Observer
- **TicketController.java** - Pont entre vues et modèle métier. Convertit Domain ↔ DTO, orchestre les opérations CRUD
- **TicketStateListener.java** - Interface Observer pour notifications de changements d'état

`models/` - Objets de transfert de données

- **TicketDTO.java** - Données immuables de ticket pour affichage (ID, titre, statut, priorité, créateur, assigné, description, date)
- **UserDTO.java** - Données utilisateur (ID, nom, rôle, isAdmin)
- **ContentItemDTO.java** - Item de contenu découplé du domaine (type: TEXT/IMAGE/VIDEO, data, metadata)
- **TicketTableModel.java** - Modèle de table Swing pour afficher les tickets

`views/` - Interfaces graphiques

- **TicketManagerGUI.java** - Fenêtre principale avec table des tickets et toolbar d'actions. Implémente `TicketStateListener`

`views/dialogs/`

- **LoginDialog.java** - Dialogue de sélection d'utilisateur
- **CreateTicketDialog.java** - Dialogue de création de ticket avec ContentBuilderPanel
- **EditTicketDialog.java** - Dialogue de modification de ticket
- **ContentBuilderPanel.java** - Panel pour construire du contenu composite (texte + images + vidéos)

`views/components/`

- **TicketDetailPanel.java** - Composant d'affichage formaté des détails de ticket

`services/` - Services métier transverses

- **FileService.java** - Gestion I/O (génération noms de fichiers, sauvegarde de rapports)

- **PermissionService.java** - Vérification centralisée des permissions (CREATE_TICKET, EDIT_TICKET, ASSIGN_TICKET, etc.)

`validators/` - Validation

- **TicketValidator.java** - Validation centralisée des données de tickets (titre, priorité). Retourne `ValidationResult` (pattern Result Object)

`utils/` - Utilitaires

- **ErrorHandler.java** - Gestion uniforme des messages utilisateur (showUserError, showTechnicalError, showWarning, showSuccess, confirm)

Flux de Données MVC

Exemple: Création d'un ticket

1. **View** - `CreateTicketDialog` collecte titre, priorité et `List<ContentItemDTO>` via `ContentBuilderPanel`
2. **Validation** - `TicketValidator.validateTitle()` et `validatePriority()`
3. **Controller** - `TicketController.createTicketWithContentItems()` convertit les DTOs en objets Domain (`Content`)
4. **Model** - `ApplicationState.addTicket()` stocke le ticket et notifie les observers
5. **Observer** - `TicketManagerGUI.onTicketsChanged()` rafraîchit automatiquement l'affichage

---

Principes Architecturaux

Pattern MVC

- **Model**: DTOs immuables + ApplicationState (pas de dépendance Swing)
- **View**: Composants Swing utilisant uniquement des DTOs (aucune création d'objets Domain)
- **Controller**: TicketController fait toutes les conversions Domain ↔ DTO

Pattern Observer

- `ApplicationState` notifie `TicketStateListener` lors de changements
- Rafraîchissements automatiques de la GUI (pas d'appels manuels `loadTickets()`)

Injection de Dépendances

- Tous les dialogues reçoivent `TicketController` par constructeur
- Même instance partagée, testable avec mocks

Services Stateless

- FileService, PermissionService, TicketValidator, ErrorHandler: méthodes `static`, thread-safe, réutilisables