Cours 05 - C++ pour les mathématiques appliquées

Conception Objet
De l’analyse à la conception logicielle

La dernière fois…

Ce que nous avons vu en analyse

Diagrammes UML d'analyse

- Cas d'utilisation
- Séquence
- États-transitions
- Classes d'analyse

Objectif

- Comprendre le domaine métier
- Modéliser les besoins
- Valider avec le client

Aujourd'hui : Conception

- Architecture logicielle
- Principes de conception
- Patterns architecturaux
- Conception détaillée

Passage à l'implémentation

- Du modèle au code
- Choix techniques
- Structure du système

Conception logicielle

Structurer avant de coder

Conception architecturale

Qu’est-ce qu’une architecture logicielle ?

Définition

- Structure générale du système
- Constituants de **haut niveau**
- **Interactions** entre éléments

Analogie avec l’architecture

- Plans d’un bâtiment
- Structure porteuse
- Organisation des espaces
- Indépendant de la décoration

Objectifs

- Organiser le code
- Faciliter la maintenance
- Permettre l’évolution
- Répartir les responsabilités

Niveau d’abstraction

- Plus haut niveau que le code
- Vue globale du système
- Décisions structurelles
- Contraintes et choix techniques

Modèle des 4+1 vues de Kruchten

Décrire l'architecture selon 5 perspectives

💡 **Idée clé :** Différentes parties prenantes ont besoin de vues différentes du système

Vue logique

Organisation fonctionnelle du système

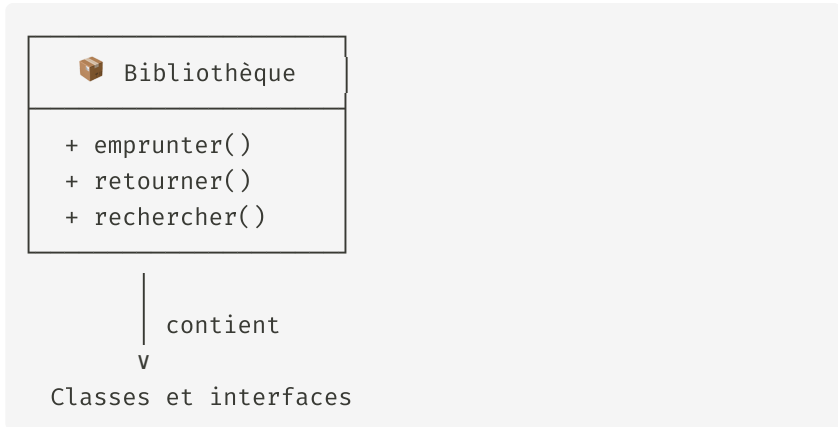
Éléments

- Sous-systèmes
- Couches (layers)
- Paquetages (packages)
- Classes et interfaces

Objectif

- Organiser les fonctionnalités
- Définir les responsabilités
- Structurer le domaine métier

Exemple : Paquetage UML



Vue de réalisation

Organisation physique des fichiers

Concerne

- Fichiers sources (.cpp, .h)
- Exécutables
- Bibliothèques (.so, .dll)
- Documentation

Gestion

- Versions (Git)
- Configurations
- Build system (CMake, Make)
- Dépendances

Exemple : Structure projet

```
projet/
├── src/
│   ├── core/
│   ├── utils/
│   └── main.cpp
├── include/
│   └── *.h
├── lib/
├── build/
├── docs/
└── CMakeLists.txt
```

Diagramme : composants UML

Vue des processus

Décomposition en exécution concurrente

Éléments

- **Processus** : programmes indépendants
- **Threads** : fils d'exécution
- **Synchronisation** : mutex, sémaphores
- **Communication** : queues, pipes

Importance

- Systèmes multi-tâches
- Applications parallèles
- Performances

Exemple : Application avec threads

```
Programme principal
     │
     ├─→ Thread UI (interface)
     │
     ├─→ Thread Calcul
     │
     └─→ Thread I/O

Synchronisation nécessaire !
```

Problématiques

- Deadlocks
- Race conditions
- Performance

Vue de déploiement

Ressources matérielles et réseau

Éléments

- Serveurs
- Machines clientes
- Liens réseau
- Bases de données

Concerne

- Performances système
- Tolérance aux pannes
- Scalabilité
- Sécurité réseau

Exemple : Application web

```
Client navigateur
        │
        │ HTTPS
        ∨
Serveur Web
        │
        │ SQL
        ∨
Base de données
```

Diagramme : déploiement UML

- Nœuds matériels
- Artéfacts logiciels
- Connexions

Principes de conception architecturale

Comment structurer intelligemment ?

Objectifs de l'architecture

Pourquoi structurer ?

Modularité

- Découper en modules
- Responsabilités claires
- Développement parallèle
- Tests unitaires faciles

Évolutivité

- Ajouter fonctionnalités
- Modifier sans tout casser
- Maintenance facilitée
- Coût réduit

Isolation

- Limiter propagation erreurs
- Changements localisés
- Interfaces stables
- Réutilisation possible

Question centrale : Comment découper le système en modules cohérents et faiblement couplés ?

Trois principes fondamentaux

Couplage, Cohésion, Protection des variations

1. Couplage faible - Degré de dépendance entre paquetages

Fort : modifications en cascade, difficile à comprendre, dur à réutiliser

Faible : indépendance, modifications locales, facile à tester

2. Cohésion forte - Cohérence des services d'un paquetage

Faible : fourre-tout, responsabilités floues

Forte : services liés, objectif clair, facile à comprendre

3. Protection des variations - Stabilité des paquetages très utilisés

Instable : beaucoup de dépendants, variations fréquentes → chaos

Stable : paquetages centraux figés rapidement

Couplage : mesure des dépendances

Définition

- Degré de liaison entre paquetages
- Nombre de dépendances

Problèmes du couplage fort

- Modifications en cascade
- Difficile à comprendre isolément
- Difficile à réutiliser
- Sensible aux changements

Objectif

- Minimiser les dépendances
- Interfaces claires et stables

Exemple

Cohésion : mesure de la cohérence

Définition

- Liens entre services du paquetage
- Unité de propos

Avantages cohésion forte

- Plus facile à comprendre
- Plus facile à réutiliser
- Plus facile à maintenir
- Moins affectée par changements

Exemples

```
 Cohésion faible : java.util

Collections, Événements,
Date/Heure, Internationalisation,
String tokenizer, Random, BitSet ...

Fourre-tout sans lien !


Cohésion forte

PackageGeometrie
  - Point, Vecteur
  - Triangle, Polygone
  - Calculs géométriques

Tout est lié à la géométrie
```

Protection des variations

Stabiliser ce qui est central

Constat

- Certaines parties évoluent vite
- D'autres doivent être stables
- Les dépendants souffrent des variations

Principe

- Paquetage utilisé par beaucoup → doit être stable
- Paquetage peu utilisé → peut varier
- Éviter variations dans paquetages centraux

Stratégie

```
Stabiliser rapidement :
  - Interfaces publiques
  - Paquetages de base
  - APIs centrales

Laisser varier :
  - Implémentations internes
  - Paquetages périphériques
  - Plugins
```

Exemple

- STL : très stable
- Classes métier : peuvent évoluer

Architecture en couches

Un pattern fondamental

Architecture en couches - Principe

Concept

- Logiciel organisé en **couches**
- Chaque couche = ensemble cohérent de classes
- Interface claire entre couches

Propriétés importantes

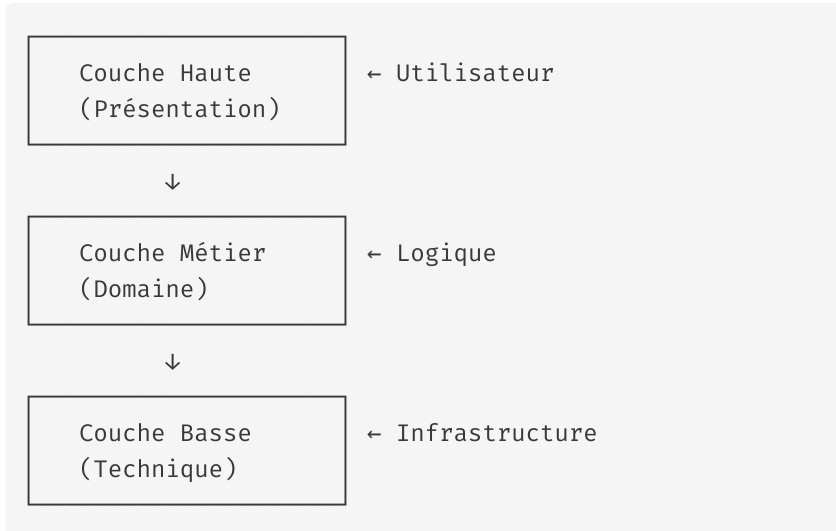
Couches ordonnées

- Couche N accède seulement à N-1, N-2…
- Flux de dépendances descendant

Couches étanches (optionnel)

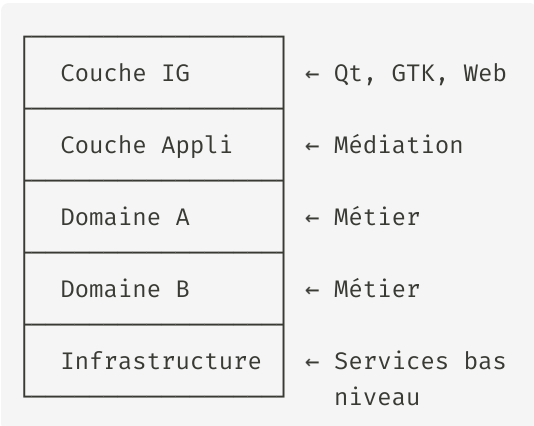
- Couche N accède **uniquement** à N-1
- Isolation stricte

Schéma générique



Exemple 1 : Application avec interface graphique

Architecture



Rôle de chaque couche

Interface Graphique (IG)

- Widgets, fenêtres
- Événements utilisateur
- Affichage

Application

- Coordination
- Médiation IG ↔ Domaine
- Contrôleurs

Domaine A, B

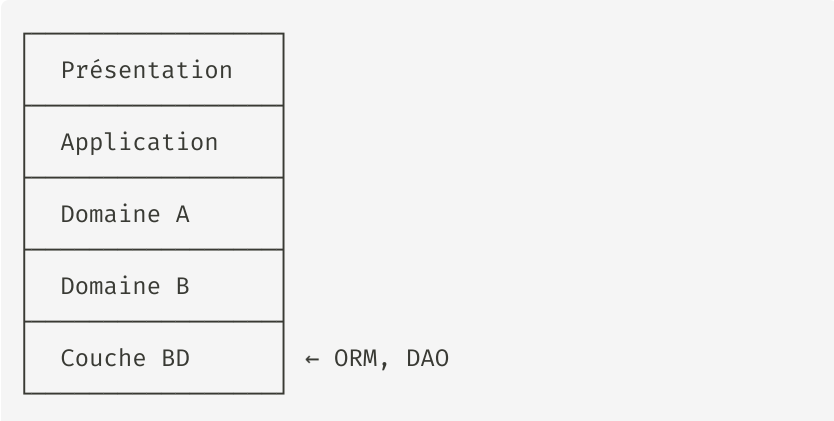
- Logique métier
- Règles de gestion
- Classes du domaine

Infrastructure

- Fichiers, réseau, BD
- Logging, configuration

Exemple 2 : Application avec base de données

Architecture



Couche BD

- Mapping objet/relationnel
- Requêtes SQL
- Transactions
- Cache

Exemple concret

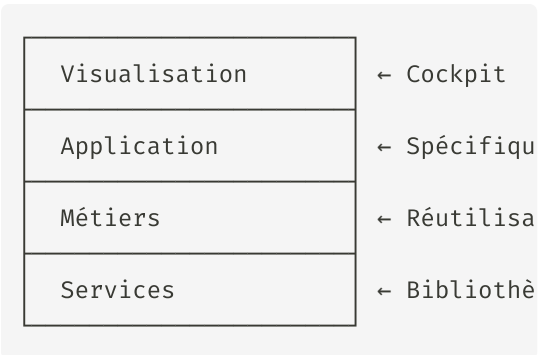
- Présentation : API REST
- Application : Endpoints
- Domaine : Classes métier
- BD : PostgreSQL + ORM

Technologies

- ORM : Object-Relational Mapping
- DAO : Data Access Object

Exemple 3 : Avionique Thales

Architecture d’un système embarqué



Couche Visualisation

- Affichage cockpit
- Interfaces pilote

Couche Application

- Spécifique à l’avion
- Configuration

Couche Métiers

- Briques réutilisables
- Navigation, radar, etc.

Couche Services

- Filtres, calculs
- Utilitaires génériques

Avantages de l’architecture en couches

1. Maintenance facilitée

- Modification d’une couche → pas d’impact sur inférieures
- Si interface stable → pas d’impact sur supérieures
- Isolation des changements

2. Réutilisation

- Couches domaine communes à plusieurs applis
- Bibliothèques partagées
- Composants standards

3. Portabilité

- Dépendances OS dans couches basses
- Couches hautes indépendantes
- Portage = réécrire couches basses

4. Tests

- Tests par couche
- Mocks pour couches inférieures
- Tests d’intégration progressifs

Inconvénient : Problème d'efficacité si trop de couches étanches (overhead d'appels)

Architecture MVC

Modèle - Vue - Contrôleur

MVC - Généralités

Historique

- Introduit dans Smalltalk (1980)
- Pattern fondamental pour IHM
- Largement utilisé aujourd'hui

Principe

- Séparer 3 responsabilités :
  - **Modèle** : données et logique
  - **Vue** : affichage
  - **Contrôleur** : entrées utilisateur

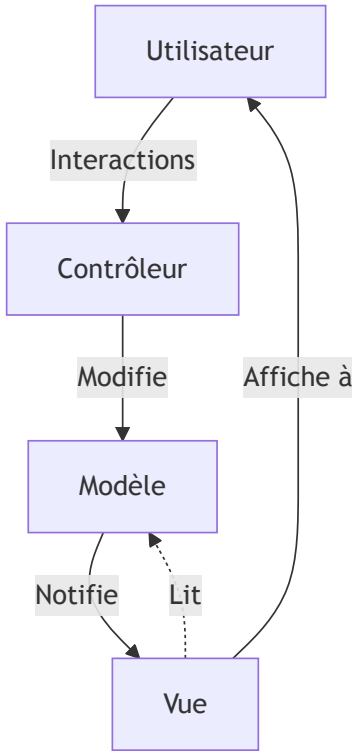
Pourquoi MVC ?

- Interface graphique
- Données manipulées
- Contrôle centralisé

Applications

- Applications desktop
- Applications web
- Frameworks (Spring, Django, Rails)

Schéma MVC classique



Flux d'interactions

1. Utilisateur → Contrôleur
   - Clics, saisies
2. Contrôleur → Modèle
   - Actions, commandes
3. Modèle → Vue
   - Notifications de changements
4. Vue → Utilisateur
   - Mise à jour affichage

Pattern Observer

- Vue observe le Modèle
- Mise à jour automatique

Modèle - Responsabilités

Rôle du Modèle

- Classes métier principales
- Données + logique
- Règles de gestion

Caractéristiques

- Indépendant Vue/Contrôleur
- Notifie changements d'état

Exemple : Catalogue

```
class Catalogue {
    vector<Document*> documents;
    vector<Observer*> vues;

    void ajouter(Document* doc) {
        documents.push_back(doc);
        notifierVues();
    }

    vector<Document*>
    rechercher(string titre) {
        // Recherche
        return resultats;
    }

    void notifierVues() {
        for(auto v : vues)
            v->update();
    }
};
```

Vue et Contrôleur

Vue

Affichage des données

```
class VueCatalogue : Observer {
    Catalogue* catalogue;

    void update() {
        auto docs =
            catalogue->getDocuments();
        afficherListe(docs);
    }
};
```

Observe le Modèle, se met à jour auto

Contrôleur

Gère les événements

```
class ControleurBiblio {
    Catalogue* catalogue;

    void onAjouter(Document* d) {
        catalogue->ajouter(d);
    }

    void onRecherche(string t) {
        auto res =
            catalogue->rechercher(t);
        vue->afficher(res);
    }
};
```

Traduit actions → commandes

Avantages de MVC

Vues multiples

- Plusieurs vues du même modèle
- Vue liste + vue détaillée
- Synchronisation automatique

Portabilité

- Modèle indépendant de l'IHM
- Changement d'IHM facile
- Code métier réutilisable

Évolution

- Ajout de fonctionnalités IHM facile
- Menu, boutons, raccourcis
- Personnalisation à l'exécution

Exemples concrets

Système de bibliothèque

- Modèle : Catalogue, Emprunts
- Vues : Liste documents, Statistiques
- Contrôleur : Recherche, Emprunt

Application web moderne

- Modèle : Backend (API)
- Vues : React, mobile app
- Contrôleur : Routes, endpoints

Tests facilités

- Modèle testable sans IHM
- Vues mockées
- Contrôleurs isolés

Variante MVC pour le web

Architecture 3-tiers

Flux modifié

```
Vue (Frontend)
      ↕
Contrôleur (Backend)
      ↕
Modèle (Base de données)
```

Différences

- Vue envoie au Contrôleur
- Contrôleur modifie Modèle
- Contrôleur renvoie à Vue
- Pas d'accès direct Vue ↔ Modèle

Frameworks

- **PHP** : Symfony, Laravel
- **Java** : Spring MVC, Struts
- **Python** : Django, Flask
- **JavaScript** : Angular, Vue.js

Architecture REST

- Contrôleur = API endpoints
- Modèle = ORM + BD
- Vue = SPA (Single Page App)

Conception objet détaillée

Du modèle d'analyse au code

Généralités de conception

Objectif de cette étape

Conception détaillée

- Diagramme de classes de conception
- Classes logicielles (pas conceptuelles)
- Inspiré du diagramme d'analyse
- Prêt pour l'implémentation

Évolutions par rapport à l'analyse

- Ajout de classes techniques
- Ajout d'attributs et méthodes
- Modification d'associations
- Suppression si nécessaire

Résultat

- Peut être traduit en code
- Langage objet (C++, Java…)
- Structure complète
- Interfaces définies

Activités

- Affectation des responsabilités
- Définition des opérations
- Choix des structures de données
- Gestion des erreurs

Affectation des responsabilités

Types de responsabilités

1. Connaissances

- Données encapsulées
- Objets connexes
- Éléments calculables

2. Comportements

- Réaliser une action
- Créer des objets
- Coordonner activités

Exemple : Emprunt

```
class Emprunt {
    Utilisateur* utilisateur;
    Exemplaire* exemplaire;
    Date dateRetourPrevue;

    // Comportement
    double calculerPenalite() {
        int jours = Date::now()
                    - dateRetourPrevue;
        return (jours > 0) ? jours * 0.50 : 0;
    }

    // Coordonner
    void retourner() {
        exemplaire->liberer();
        utilisateur->libererQuota();
    }
};
```

Principe de l'Expert

Principe

- Tâche effectuée par l'objet qui a l'information nécessaire

Avantages

- Encapsulation respectée
- Couplage faible
- Code intuitif

Exemple

```
// Qui calcule les pénalités ?

class Emprunt {
    Date dateRetourPrevue;

    // Expert : a la date
    double calculerPenalite() {
        int jours = Date::now()
                    - dateRetourPrevue;
        return max(0, jours) * 0.50;
    }
};

//  Pas dans Utilisateur
//  Pas dans classe utilitaire
```

Principe du Créateur

Principe

- B crée A si :
  - B contient A
  - B agrège A
  - B a les infos pour créer A

Objectif

- Créateur connecté à l'objet
- Dépendances naturelles

Exemple

```
class Utilisateur {
    vector<Emprunt*> emprunts;

    // Créateur naturel
    void emprunter(Exemplaire* ex) {
        if (quotaAtteint())
            throw QuotaException();

        Date retour = Date::now() + 14;
        Emprunt* e = new Emprunt(
            this, ex, retour);
        emprunts.push_back(e);
    }
};

// Utilisateur contient emprunts
// A l'info pour créer
```

Principe de faible couplage

Couplage = dépendances

Fort

- Beaucoup de dépendances
- Difficile à réutiliser
- Sensible aux changements

Faible

- Peu de dépendances
- Facile à réutiliser
- Robuste

Réduction par interfaces

```
//  Fort
class Service {
    MySQLDatabase* db;
    FileLogger* log;
    EmailSender* email;
    // ... 10 autres
};

// Faible
class Service {
    Database* db;  // Interface
    Logger* log;   // Interface
    Notifier* not; // Interface
};
```

Dépendre d'abstractions

Principe de forte cohésion

Cohésion = liens entre tâches

Faible cohésion

- Tâches sans rapport
- Fourre-tout
- Difficile à maintenir

Forte cohésion

- Tâches liées
- Objectif clair
- Facile à comprendre

Exemples

```
//  Faible
class Utilitaires {
    void envoyerEmail();
    void racineCarree();
    void triRapide();
};

// Forte
class EmailService {
    void envoyer();
    void valider();
    void formater();
};
```

Une classe = un objectif clair

Principe du Contrôleur

Principe

- Événements externes → classes de contrôle
- Séparer logique de l'interface

Types

1. Contrôleur façade

- Système global
- Point d'entrée unique

2. Contrôleur cas d'usage

- Un par fonctionnalité
- Coordonne objets

Exemple

```
// Vue
class BoutonSauvegarder {
    void onClick() {
        // Déléguer
        controleur.sauvegarder();
    }
};

// Contrôleur
class AppControleur {
    Document* doc;

    void sauvegarder() {
        if (doc→estValide()) {
            doc→sauvegarder();
            vue→message("OK");
        }
    }
};
```

Logique dans Contrôleur, pas dans Vue

Diagrammes UML de conception

Documenter les choix

Diagramme de classes logicielles

Évolution du diagramme d'analyse

Ajouts possibles

- Classes techniques
- Classes utilitaires
- Collections, iterateurs
- Gestionnaires

Modifications

- Attributs détaillés (types)
- **Opérations** (signatures complètes)
- Visibilité (+, -, #)
- Associations précisées

Exemple : de l'analyse à la conception

Analyse

```
Commande
————————
- date
- total
```

Conception

```
Commande
————————
- date: Date
- lignes: vector<Ligne*>
- client: Client*
————————
+ ajouterLigne(Produit*, int): void
+ calculerTotal(): double
+ valider(): bool
- verifierStock(): bool
```

Exemple : Système de bibliothèque - Description

Application complète de gestion

Un système de gestion de bibliothèque universitaire avec interface graphique.

Composants principaux :

- **Catalogue** : collection de documents (livres, revues, thèses)
- **Utilisateurs** : étudiants (quota 3) et enseignants (quota 5)
- **Emprunts** : gestion avec dates et pénalités
- **Interface** : application Qt avec vues multiples

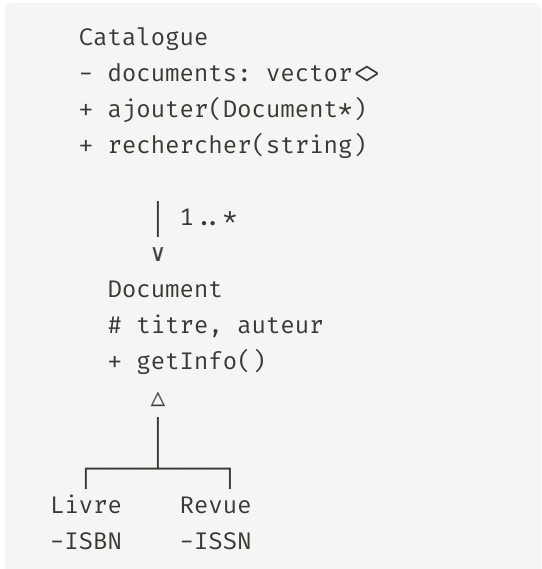
Fonctionnalités :

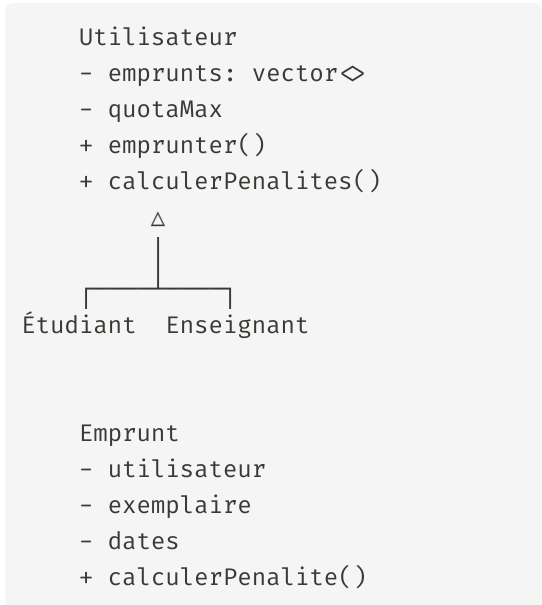
- Recherche de documents (titre, auteur, ISBN)
- Emprunt et retour de documents
- Calcul automatique des pénalités (0.50€/jour)
- Statistiques et historique
- Gestion du catalogue par le bibliothécaire

Contraintes :

- Base de données PostgreSQL
- Synchronisation temps réel entre vues
- Architecture MVC

Exemple : Diagramme de classes de conception





Évolutions vs analyse

Ajouts

- Classes techniques
- Types précis
- Opérations complètes
- Visibilité (+, -, #)

Exemples

- `ControleurBiblio`
- `VueCatalogue : Observer`
- `vector<Document*>`
- `calculerPenalites(): double`

Précisions

- Multiplicités exactes
- Navigation
- Dépendances

Conclusion

Points clés du cours

Architecture

- 4+1 vues de Kruchten
- Architecture en couches
- Pattern MVC
- Paquetages UML

Principes

- Couplage faible ⬇️
- Cohésion forte ⬆️
- Protection des variations 🛡️
- Expert, Créateur, Contrôleur

Conception détaillée

- Affectation des responsabilités
- Diagrammes de classes logicielles
- Séquences de conception
- Cohérence entre diagrammes

Objectif final

- Code bien structuré
- Maintenable et évolutif
- Respecte les principes
- Prêt pour l'implémentation

La prochaine fois

Patrons de conception (Design Patterns)

Patterns créationnels

- Singleton
- Factory
- Builder

Patterns structurels

- Adapter
- Decorator
- Composite

Patterns comportementaux

- Observer
- Strategy
- Iterator

💡 **Préparez-vous :** Les patterns sont des solutions éprouvées à des problèmes récurrents