25/10/2025

**Rapport Projet AP4A**

**Automne 2025**

**Simulation d’un système de badges et lecteurs de badges**

Journet Célia

Table of Contents

[**INTRODUCTION** 1](#_Toc212468520)

[**Besoins du Projet** 2](#_Toc212468521)

[**Description Fonctionnelle** 2](#_Toc212468522)

[Liste des fonctionnalités exigées 2](#_Toc212468523)

[Spécialisation des classes, d’après le sujet : 3](#_Toc212468524)

[**Règles de Gestion** 6](#_Toc212468525)

[**Conception du Projet** 8](#_Toc212468526)

[**Diagramme de classes UML** 8](#_Toc212468527)

[Hiérarchie principale des classes : 8](#_Toc212468528)

[Classes de gestion : 10](#_Toc212468529)

[Relations et associations clés : 10](#_Toc212468530)

[**Annexe** 12](#_Toc212468531)

# **INTRODUCTION**

La sécurisation des accès dans un campus universitaire, où un grand nombre de personnes aux accès variés se mélangent, est une nécessité. L'objectif était de concevoir et développer en C++ un simulateur de contrôle d'accès modélisant un système de badges et de lecteurs de badges.

Ce projet m’a permis de mettre en application les principes de la programmation orientée objet (POO) pour créer une architecture modulaire, simulant les interactions entre un serveur central, des lecteurs de badges, des badges et des utilisateurs, le tout centralisé dans un ordonnanceur. Les défis principaux ont résidé dans la modélisation fine des droits d'accès et dans l'ordonnancement des événements sur une journée type. Les résultats des transactions sont tracés via une console et enregistrés dans des fichiers de logs.

Ce rapport détaillera dans un premier temps l'analyse et la conception du système. Puis, nous présenterons les choix d'implémentation des classes principales et les algorithmes de gestion des accès. Enfin, nous validerons le fonctionnement du simulateur à travers ses résultats et proposerons une analyse critique ainsi que des pistes d'amélioration.

# **Besoins du Projet**

## **Description Fonctionnelle**

### Liste des fonctionnalités exigées

**Fonctionnalités principales :**

* Gestion des**personnels** (étudiants, enseignants, sécurité, etc.) avec leurs**badges nominatifs**
* Gestion des**lecteurs de badges**associés à des portes (bâtiments, salles)
* **Contrôle d'accès**avec droits variables selon le type de personnel
* **Simulation temporelle**d'une journée type via un ordonnanceur

**Traitement des accès :**

* **Workflow :** Badge présenté → Lecteur → Serveur → Vérification droits → Autorisation/Refus
* **Service central :**demandeAcces(Badge, LecteurBadge) avec validation des droits
* **Configuration**des droits via fichier chargé au démarrage

**Contraintes techniques :**

* **Langage :** C++ orienté objet
* **Communication**par appels de méthodes
* **Compilation multi-plateforme**(CMakeLists.txt ou .pro)
* **Mécanismes C++ avancés :** héritage, polymorphisme, STL, etc.

**Sorties et visualisation :**

* **Journalisation :** Fichier de logs avec les demandes et résultat
* **Visualisation console**
* **Informations tracées :** heure, nom, prénom, statut, type et lieu du lecteur, résultat

### Spécialisation des classes, d’après le sujet :

N’ayant pas de diagramme UML de conception, la structure détaillée des classes et leurs responsabilités sont décrites ci-dessous.

**Classe Badge :**

* Chaque badge doit être **nominatif** et associé de manière unique à une personne
* Le badge doit contenir les **informations d'identification** de la personne (nom, statut, identifiant unique)
* Le badge doit permettre de **récupérer les droits d'accès** de la personne via des méthodes appropriées
* Les badges peuvent être organisés selon une **hiérarchie d'héritage** selon le type d'accès autorisé
* Le badge doit pouvoir **initier une demande d'accès** lorsqu'il est présenté sur un lecteur

**Classe Lecteur de Badge :**

* Les lecteurs doivent être **associés à des portes spécifiques** (bâtiments ou salles de travail)
* La classe LecteurBadge doit servir de **classe mère** pour différents types de lecteurs
* Les lecteurs doivent pouvoir **communiquer avec le serveur** pour transmettre les demandes d'accès
* Chaque lecteur doit avoir une **localisation spécifique** (bâtiment, salle, zone)
* Les lecteurs doivent pouvoir **détecter la présentation d'un badge** et initier la procédure de vérification

**Classe Serveur :**

**Rôle principal :**

* **Cœur du système** de contrôle d'accès
* **Centralise** la gestion des droits et autorisations
* **Coordonne** la communication entre badges et lecteurs

**Fonctionnalités clés :**

* **Service central** : demandeAcces(Badge, LecteurBadge) pour traiter toutes les requêtes
* **Vérification des droits** selon le type de personnel et la localisation
* **Gestion de configuration** via fichier chargé au démarrage
* **Décision d'accès** : autorisation ou refus avec justification

**Traitement des demandes :**

* **Réception** des demandes depuis les lecteurs
* **Analyse** des informations badge + lecteur
* **Validation** des droits d'accès
* **Retour** de la décision au lecteur
* **Journalisation** systématique du résultat

**Caractéristiques techniques :**

* **Mode passif** : répond aux requêtes externes
* **Utilise le polymorphisme** pour gérer les différents types
* **Connaissance complète** des objets du système

**Classe Scheduler - Description Fonctionnelle**

**Rôle principal :**

* **Moteur de la simulation** temporelle
* **Ordonnance** le déroulement des événements
* **Anime** l'ensemble du système sur une base de temps

**Fonctionnalités clés :**

* **Gestion du temps** simulation d'une "journée type"
* **Séquencement** des demandes d'accès
* **Activation** des processus aux moments appropriés
* **Coordination** globale de la simulation

**Processus de simulation :**

* **Initialisation** de la séquence temporelle
* **Déclenchement** des présentations de badges
* **Organisation** du flux d'événements
* **Gestion** de la fréquence et répartition des accès

**Caractéristiques techniques :**

* **Boucle d'ordonnancement** principale
* **Base de temps** prédéfinie et configurable
* **Séquence d'appels** coordonnée entre objets
* **Flexibilité** dans l'organisation des appels

La différence de développement entre Badge/Lecteur de Badge et Serveur/Scheduler s’explique par le fait que le Serveur et le Scheduler sont des éléments centraux et nécessitent une logique métier élaborée tandis que les Badges et Lecteurs ont des rôles plus spécialisés et limités

## **Règles de Gestion**

Voici toutes les règles émises par le sujet. Ici nous ne détaillerons pas ce qui a été fait ou non, ceci sera fait plus tard.

**Gestion des droits d'accès :**

* Les droits d'accès et priorités diffèrent selon le type de personnel
* Un même personnel peut disposer de droits d'accès multiples
* La configuration des droits d'accès est centralisée au niveau du serveur
* Les droits doivent pouvoir être chargés via un fichier de description au démarrage

**Hiérarchie des personnels :**

* **Étudiants** : droits d'accès basiques selon leur filière
* **Enseignants** : droits étendus, pouvant inclure des accès de type sécurité
* **Personnel de sécurité** : droits étendus avec priorités élevées
* **Personnel administratif** : droits limités aux zones administratives
* **Enseignants-chercheurs** : droits combinés (enseignement + accès spécifique au laboratoire)

**Gestion des priorités :**

* Les personnels de sécurité disposent de la priorité la plus élevée
* Les enseignants et enseignants-chercheurs ont une priorité standard
* Les étudiants et personnels administratifs ont une priorité basique

**Processus d'accès :**

* Une demande d'accès est initiée par le scan d'un badge sur un lecteur
* Le serveur analyse chaque demande et vérifie les droits correspondants
* Les autorisations d'accès sont accordées ou refusées en fonction des droits
* Les portes d'entrée s'ouvrent uniquement après autorisation du serveur

**Journalisation :**

* Toutes les tentatives d'accès doivent être journalisées
* Les accès autorisés et les tentatives infructueuses sont enregistrés dans des fichiers séparés
* Les logs doivent contenir : heure d'accès, nom de la personne, statut, résultat

# **Conception du Projet**

## A screenshot of a computer screen AI-generated content may be incorrect.**Diagramme de classes UML**

Figure 1 : Diagramme de classes UML

Le diagramme pourra être retrouvé dans le dossier envoyé sous le nom : « UML\_AP4A\_TP1B\_Journet\_Celia.png ».

Notre conception repose sur une architecture orientée objet organisée autour de plusieurs hiérarchies de classes complémentaires. Le diagramme UML présente la structure complète du système de contrôle d'accès.

### Hiérarchie principale des classes :

**Classe abstraite**Personne :

La classe abstraite Personne représente l'entité centrale du système. Chaque personne est caractérisée par son nom et son prénom et ses statuts. Cette classe permet une gestion dynamique des statuts via les méthodes ajouterStatut() et supprimerStatut(), et permet de vérifier l'appartenance à un statut via aStatut(). La méthode virtuelle pure getStatutPrincipal() = 0 impose aux classes dérivées de définir leur statut principal, garantissant ainsi le polymorphisme nécessaire au système d'accès.

**Classes dérivées de**Personne :

Les classes dérivées (publiquement) de Personnes sont : Eleve, Professeur, Securite, Admin, Chercheur. Chacune de ces classes redéfinissent getStatutPrincipal() selon le statut imposé.

**Classe abstraite**Badge :

La classe abstraite Badge représente les badges que possède toutes les personnes du campus universitaire. Elle associe une Personne à un identifiant unique. Il est possible de contrôler la validité de chaque badge avec l’attribut « validite » ainsi que la fonction isActif(). La méthode virtuelle pure getStatutUtilisateur() = 0 oblige les classes dérivées à renvoyer le statut principal de la Personne.

**Classe Dérivée Template de** Badge:

Pour les classes dérivées de Badge, nous avons décidé de créer une classe template fonctionnant pour EleveBadge(), ProfesseurBadge(), SecuriteBadge(), AdminBadge(), ChercheurBadge(). Les badges templates assurent qu'un badge créé pour un type spécifique (ex: Eleve) ne peut être associé qu'à ce type, garantissant ainsi la cohérence du système.

**Classe abstraite**LecteurBadge :

La classe abstraite LecteurBadge représente les points d’accès du campus, représentaient par une localisation et un type de porte spécifique. La classe LecteurBadge possède une référence sur l’unique Serveur du système pour la vérification des droits et l’accès des personnes. La méthode virtuelle pure getTypePorte() = 0 impose aux classes dérivées le renvoie d’un type de porte différent, garantissant ainsi le polymorphisme nécessaire au système d'accès, tout comme les classes abstraites Personne et Badge.

**Classes dérivées de** LecteurBadge:

Les classes dérivées (publiquement) de LecteurBadge sont : LectBadgeBat, LectBadgeLab, LectBadgeClasse, LectBadgeZoneSecu, LectBadgeZoneAdmin, LectBadgeBibli, LectBadgeProf, LectBadgeCafet. Chacune de ces classes redéfinissent getTypePorte() selon le type de la porte.

### Classes de gestion :

**Classe**Serveur :

La classe Serveur est le cœur du système de vérification des accès. Les droits sont centralisés via un map de configuration (map<String, map<String>, bool>), rempli grâce à la fonction loadConfiguration(). Pour le traitement des demandes d’accès le serveur utilise la méthode askAcces(Badge&, LecteurBadge&) qui va alors appeler la méthode privée du Serveur verifierTotalAcces(Badge&, LecteurBadge&) et ainsi renvoyer dans les logs et la console le résultat de la demande d’accès. La journalisation des demandes d’accès avec l’heure, la personne, la localisation, le type de porte et le résultat de la demande est stockée dans un unique fichier de log.

**Classe**Scheduler :

La classe Scheduler est un ordonnanceur de la simulation. Les lecteurs de badge ainsi que les badges sont créés et utilisés par le Scheduler (via des vecteurs) dans la fonction initialisation(). Le Scheduler simule une journée type entre 7h30 et 19h, avec une gestion d’horaires mis à jour toutes les secondes réelle, soit 15 minutes dans la simulation. Chaque créneau horaire possède une probabilité d’évènement à générer (demande d’accès) dans la fonction simulation() qui va ensuite appeler la fonction genererEvenementAleatoire(). Chaque réponse de demande d’accès est envoyé au Serveur via la fonction ecrireLog(string&, bool).

**Structure**ResultatVerification :

La structure ResultatVerification permet d’encapsuler le résultat d’une vérification d’accès. Elle continent un booléen d’autorisation ainsi que, en cas de refus, la raison. Cette structure est utilisée par le Serveur pour standardiser les réponses via la fonction.

### Relations et associations clés :

L'architecture du système repose sur plusieurs types de relations bien définies qui structurent les interactions entre les composants.

**Héritage : Spécialisation et Polymorphisme**

Le système utilise deux hiérarchies d'héritage principales :

* **Personne → Profils métier :** La classe abstraite Personne définit l'interface commune pour tous les types d'utilisateurs. Les classes Eleve, Professeur, Securite, Admin et Chercheur spécialisent cette base en redéfinissant la méthode getStatutPrincipal(), permettant un traitement polymorphique des différents profils.
* **LecteurBadge → Types de lecteurs :** La classe abstraite LecteurBadge établit le comportement commun pour tous les points d'accès. Les classes dérivées (LectBadgeBat, LectBadgeLab, LectBadgeClasse, etc…) spécialisent le type de porte via la redéfinition de getTypePorte(), adaptant le comportement à chaque contexte d'accès.

**Composition : Gestion du cycle de vie**

La classe Scheduler utilise des relations de composition pour orchestrer l'ensemble du système :

* **Conteneurs d'objets :** Le Scheduler maintient des collections de vector<LecteurBadge\*>, vector<Badge\*> assumant leur gestion pendant la simulation.
* **Coordination centrale : L**e Scheduler est donc le coordinateur principal, responsable de l'initialisation, de l'ordonnancement et de la libération des ressources.

**Association : Collaboration entre composants**

Des associations bidirectionnelles permettent la collaboration entre entités métier :

* **Badge → Personne :** Chaque badge référence la Personne à laquelle il est attribué, établissant un lien direct entre le support physique et l'identité numérique.
* **LecteurBadge → Serveur :** Les lecteurs maintiennent une référence vers le Serveur pour déléguer la vérification des droits d'accès, implémentant ainsi la séparation des préoccupations.

**Utilisation : Échanges de données**

Le pattern d'utilisation permet des interactions temporaires et ciblées :

* **Serveur → ResultatVerification :** Le serveur utilise la structure ResultatVerification pour encapsuler et transporter les résultats des vérifications d'accès, standardisant ainsi le format de réponse sans créer de dépendance permanente.

Ces relations bien définies assurent une architecture modulaire, maintenable et extensible, où chaque composant joue un rôle précis et des interactions clairement délimitées.

Cette architecture modulaire permet une extensibilité aisée pour l'ajout de nouveaux types de personnels, badges ou lecteurs, tout en maintenant une séparation claire des responsabilités.

## **Choix d’architecture**

L’architecture repose sur plusieurs décisions fondamentales qui structurent l'ensemble du système.

La séparation entre Personne et Badge permet une modélisation plus réaliste, avec une possibilité pour une même personne d’avoir un badge mais plusieurs statuts. Ce choix permet d’être plus réaliste respecte le principe de responsabilité unique et offre une flexibilité pour d'éventuelles évolutions du système.

Concernant la hiérarchie Personne, nous avons conçu une architecture d'héritage complète avec la classe abstraite Personne et ses spécialisations (Eleve, Professeur, Securite, etc...). Cependant, lors de l'implémentation, nous avons constaté une redondance fonctionnelle : la méthode getStatutPrincipal() des classes dérivées de Personne et getStatutUtilisateur() des templates de Badge produisaient la même information. Pour cette version du simulateur, nous avons donc privilégié l'approche par les badges templates, qui s'est avérée suffisante pour les besoins actuels, tout en conservant l'architecture Personne pour une éventuelle évolution future.

La conception des badges via des templates a été un choix stratégique, permettant d’apprendre à créer des fonctions templates, ainsi qu’à les utiliser. De plus, ceci a permis d’éviter la redondance de code ce qui est bien plus pratique. Les TypeBadge<T> garantissent une correspondance stricte des types à la compilation tout en évitant la duplication de code. Cette solution assure qu'un EleveBadge ne peut être associé qu'à un Eleve en tant que statut principal, éliminant ainsi les erreurs de type potentielles dès la phase de compilation.

La centralisation de la logique de contrôle d'accès dans le Serveur plutôt dans les lecteurs individuels assure une cohérence globale des politiques de sécurité. Le Serveur est le seul décisionnaire pour les autorisations d'accès, permettant une gestion unifiée de la configuration et une journalisation centralisée de toutes les tentatives d'accès.

L'ordonnanceur (Scheduler) occupe une position centrale dans l'architecture en tant que coordinateur de la simulation. Il maîtrise le temps simulé, détient les collections d'objets actifs et génère les interactions entre badges et lecteurs de manière réaliste. Cette centralisation du contrôle du flux permet une vision d'ensemble du déroulement de la simulation.

Enfin, la structure ResultatVerification standardise les échanges entre les composants en encapsulant de manière cohérente la décision d'accès et sa justification. Ce format uniforme facilite la vérification des accès et assure la clarté des interactions entre le Serveur et les Lecteurs.

L'ensemble de ces choix architecturaux amène vers un système modulaire, extensible et maintenable, où chaque composant assume une responsabilité claire tout en collaborant efficacement avec les autres selon des relations bien définies.

# **Implémentation et Développement**

# **Annexe**