

Projet CSC4102 : Gestion des clefs dans un hôtel

DUBOC Aurélien et ZHEN Alice

Année 2019–2020 — 5 mars 2020

Table des matières

1	Spécification	3
1.1	Diagrammes de cas d'utilisation	3
1.2	Priorités, préconditions et postconditions des cas d'utilisation	4
2	Préparation des tests de validation	5
2.1	Tables de décision des tests de validation	5
3	Conception	6
3.1	Liste des classes	6
3.2	Diagramme de classes	7
3.3	Diagrammes de séquence	8
4	Fiche des classes	10
4.1	Classe GestionClefsHotel	10
4.2	Classe Badge	10
4.3	Classe Chambre	11
5	Diagrammes de machine à états et invariants	12
6	Préparation des tests unitaires	13

1 Spécification

1.1 Diagrammes de cas d'utilisation

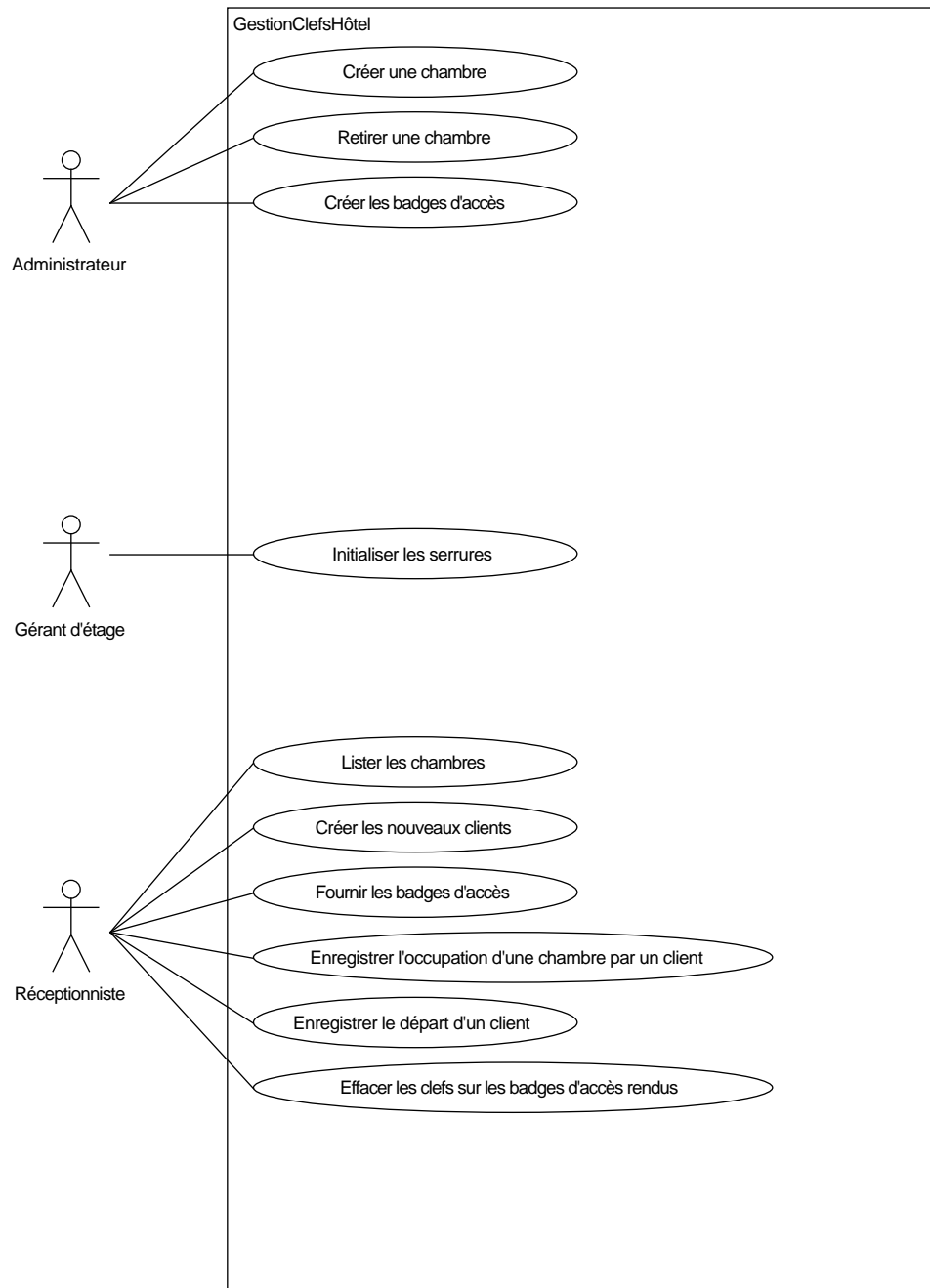


FIGURE 1 – Diagramme de cas d'utilisation

1.2 Priorités, préconditions et postconditions des cas d'utilisation

Les priorités des cas d'utilisation pour le sprint 1 sont choisies avec les règles de bon sens suivantes :

pour retirer une entité du système, elle doit y être. La priorité de l'ajout est donc supérieure ou égale à la priorité du retrait ;

pour lister les entités d'un type donné, elles doivent y être. La priorité de l'ajout est donc supérieure ou égale à la priorité du listage ;

il est *a priori* possible, c.-à-d. sans raison contraire, de démontrer la mise en œuvre d'un sous-ensemble des fonctionnalités du système, et plus particulièrement la prise en compte des principales règles de gestion, sans les retraits ou les listages.

la possibilité de lister aide au déverminage de l'application pendant les activités d'exécution des tests de validation. Par conséquent, les cas d'utilisation d'ajout sont *a priori* de priorité « haute », ceux de listage de priorité « moyenne », et ceux de retrait de priorité « basse ».

Dans la suite, nous donnons les préconditions et postconditions pour les cas d'utilisation de priorité « Haute ». Pour les autres, nous indiquons uniquement leur niveau de priorité.

HAUTE Créer une chambre

- n° 1 — précondition : identifiant/code de la chambre bien formé (non **null** et non vide) \wedge chambre avec cet identifiant inexistante \wedge graine pour la génération des clefs bien formée (non **null** et non vide)
- postcondition : chambre avec cet identifiant existante

HAUTE Créer un nouveau client

- n° 2 — précondition : client n'existe pas dans le système \wedge informations personnelles bien formées (non **null** et non vide)
- postcondition : client existant dans le système

HAUTE Créer les badges d'accès

- n° 3 — précondition : identifiant du badge bien formé (non **null** et non vide)
- postcondition : badge d'accès créé

HAUTE Enregistrer l'occupation d'une chambre par un client

- n° 4 — précondition : client existant dans le système \wedge chambre existante avec ce code \wedge chambre avec ce code inoccupée
- postcondition : client enregistré dans la chambre \wedge écriture de la nouvelle paire de clés dans le badge d'accès

Moyenne Fournir les badges d'accès

Moyenne Lister les chambres

basse Enregistrer le départ d'un client

basse Retirer une chambre

basse Effacer les clefs sur les badges d'accès rendus

2 Préparation des tests de validation

2.1 Tables de décision des tests de validation

La fiche programme du module CSC4102 ne permettant pas de développer des tests de validation couvrant l'ensemble des cas d'utilisation de l'application, les cas d'utilisation choisis sont de priorité HAUTE.

Numéro de test	1	2	3	4
Identifiant/code de la chambre bien formé (non null et non vide)	F	T	T	T
Graine pour la génération des clefs bien formée (\neq null \wedge \neq vide)		F	T	T
Chambre inexistante avec ce code			F	T
Création acceptée	F	F	F	T
Nombre de jeux de test	2	2	1	1

TABLE 1 – Cas d'utilisation « créer une chambre »

Numéro de test	1	2	3	4	5	6
Client existant dans le système	F	T	T	T	T	T
Informations du client bien formées (non null et non vide)		F	T	T	T	T
Code de la chambre bien formé (non null et non vide)			F	T	T	T
Chambre existante avec ce code				F	T	T
Chambre avec ce code inoccupée					F	T
Client enregistré dans la chambre	F	F	F	F	F	T
Nombre de jeux de test	1	2xn	2	1	1	1

TABLE 2 – Cas d'utilisation « enregistrer l'occupation d'une chambre par un client »

Numéro de test	1	2	3	4	5	6
Client existant dans le système	F	T	T	T	T	T
Informations du client bien formées (non null et non vide)		F	T	T	T	T
Code de la chambre bien formé (non null et non vide)			F	T	T	T
Chambre existante avec ce code				F	T	T
Chambre avec ce code occupée par ce client					F	T
Chambre libérée	F	F	F	F	F	T
Nombre de jeux de test	1	2xn	2	1	1	1

TABLE 3 – Cas d'utilisation « libérer une chambre »

3 Conception

3.1 Liste des classes

À la suite d'un parcours des diagrammes de cas d'utilisation et d'une relecture de l'étude de cas, voici la liste de classes avec quelques attributs :

GestionClefsHotel (la façade),

Util (classe utilitaire déjà programmée) — 'attribut de classe TAILLE_CLEF, méthodes de classe genererUneNouvelleClef et clefToString),

Clef hash

Badge identifiant, premiereClef, secondeClef

Client identifiant, nom, prénom

Chambre identifiant, premiereClef, secondeClef, graine, sel

3.2 Diagramme de classes

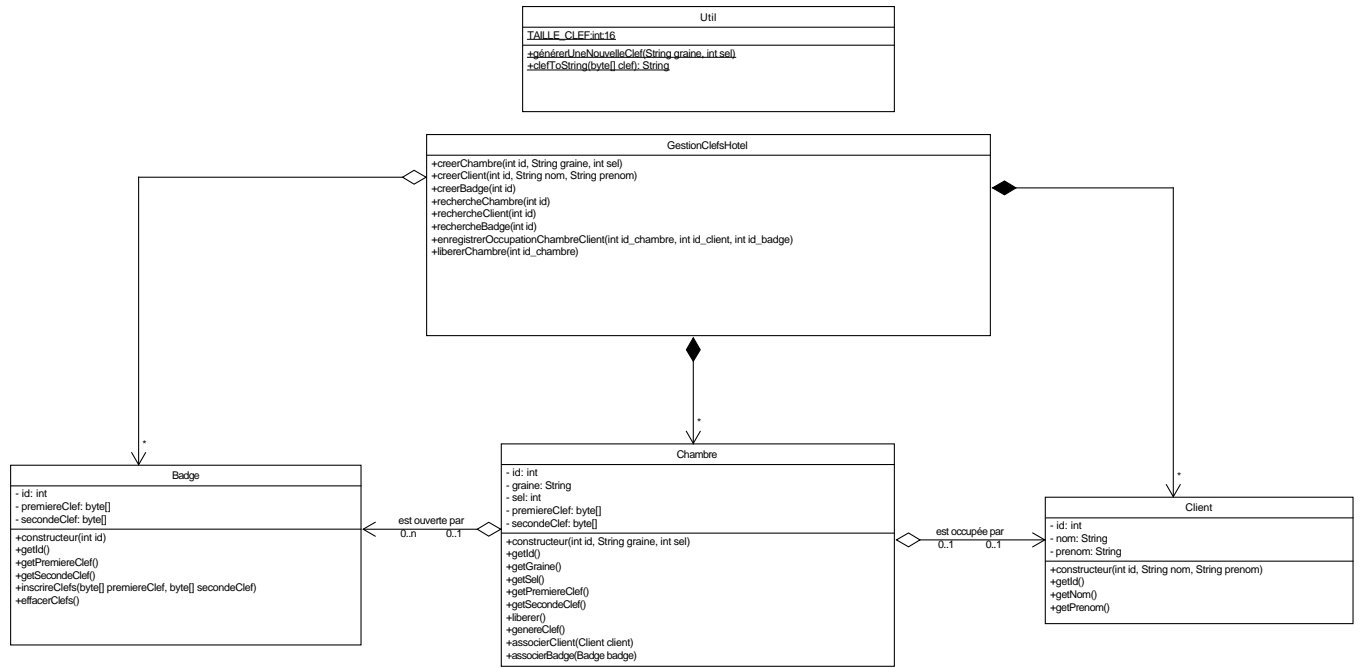


FIGURE 2 – Diagramme de classes

3.3 Diagrammes de séquence

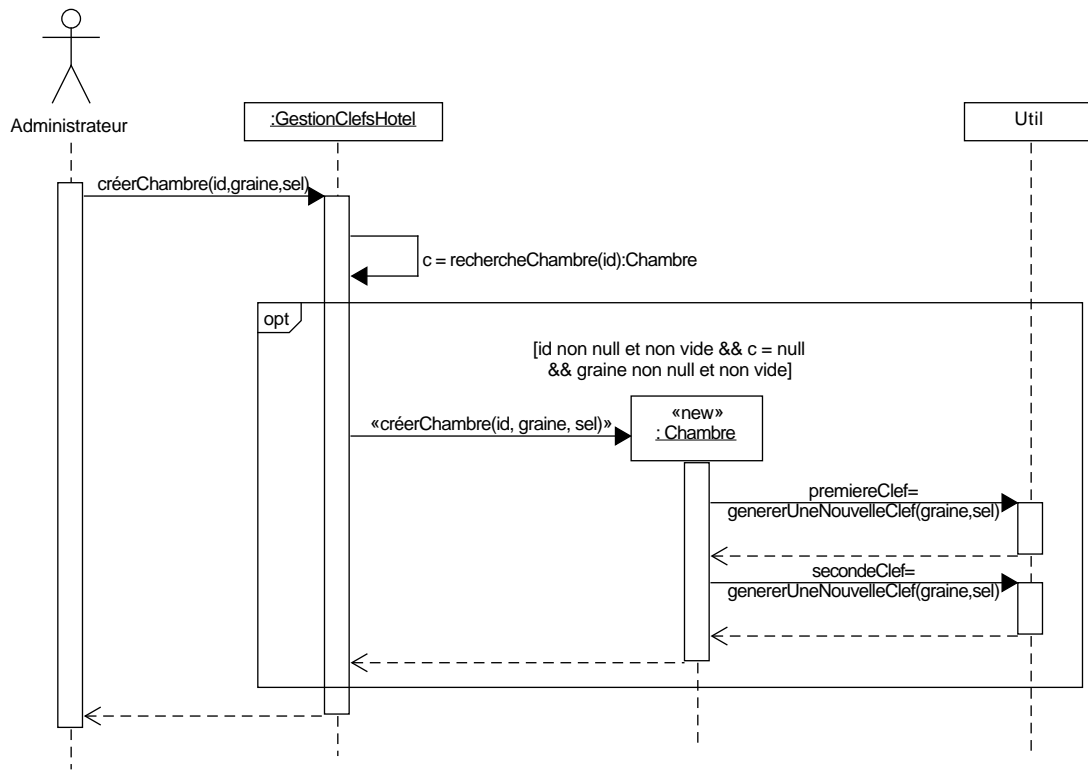


FIGURE 3 – Diagramme de séquence « créer une chambre »

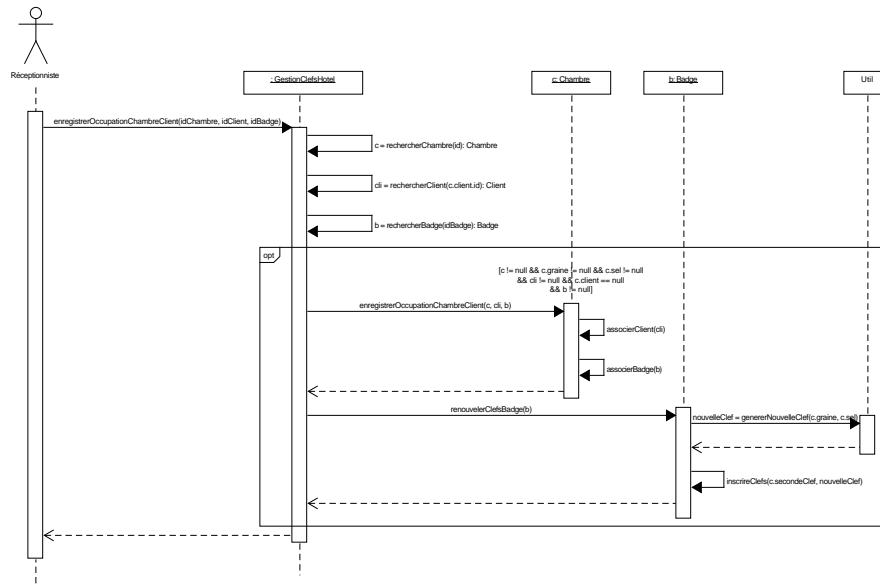


FIGURE 4 – Diagramme de séquence « enregistrer l'occupation d'une chambre par un client »

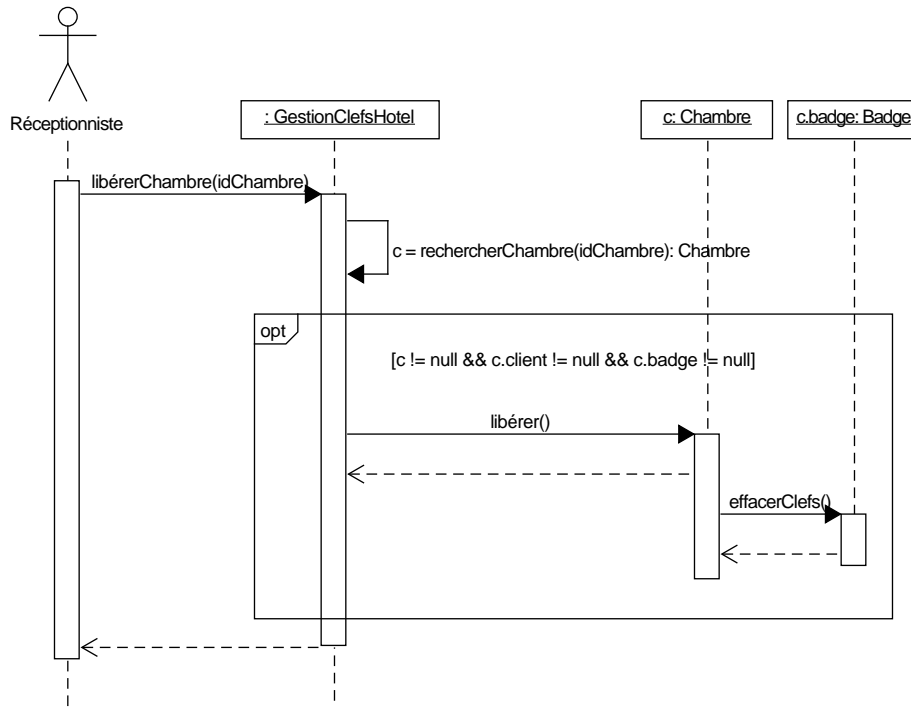


FIGURE 5 – Diagramme de séquence « libérer une chambre »

4 Fiche des classes

4.1 Classe GestionClefsHotel

GestionClefsHotel
<- attributs « association » -> - chambres : collection de @Chambre - clients : collection de @Client - badges : collection de @Badge
<- constructeur -> + GestionClefsHotel() + invariant() : booléen <- opérations « cas d'utilisation » -> + créerChambre(int id, String graine, int sel) + créerClient(int id, String nom, String prenom) + créerBadge(int id) + enregistrerOccupationChambreClient(int id_chambre, int id_client, int id_badge) + libérerChambre(int id_chambre) <- opérations de recherche -> + rechercheChambre(int id) : Chambre + rechercheClient(int id) : Client + rechercheBadge(int id) : Badge

4.2 Classe Badge

Badge
<- attributs -> - id : int - premiereClef : byte[] - secondeClef : byte[]
<- constructeur -> + Badge(int id) + destructeur() + invariant() : booléen <- opérations « cas d'utilisation » -> + getPremiereClef() : byte[] + getSecondeClef() : byte[] + inscrireClefs(byte[] premiereClef, byte[] secondeClef) + effacerClefs()

4.3 Classe Chambre

Chambre
<pre><- attributs -> - id : int - graine : String - sel : int - premiereClef : byte[] - secondeClef : byte[] <- attributs « association » -> - badge : @Badge - client : @Client</pre>
<pre><- constructeur -> + Chambre(int id, String graine, int sel) + destructeur() + invariant() : booléen <- operations « cas d'utilisation » -> + getGraine() : String + getSel() : int + getPremiereClef() : byte[] + getSecondeClef() : byte[] + associerClient(Client client) + associerBadge(Badge badge) + libérer() + genereClef() : byte[]</pre>

5 Diagrammes de machine à états et invariants

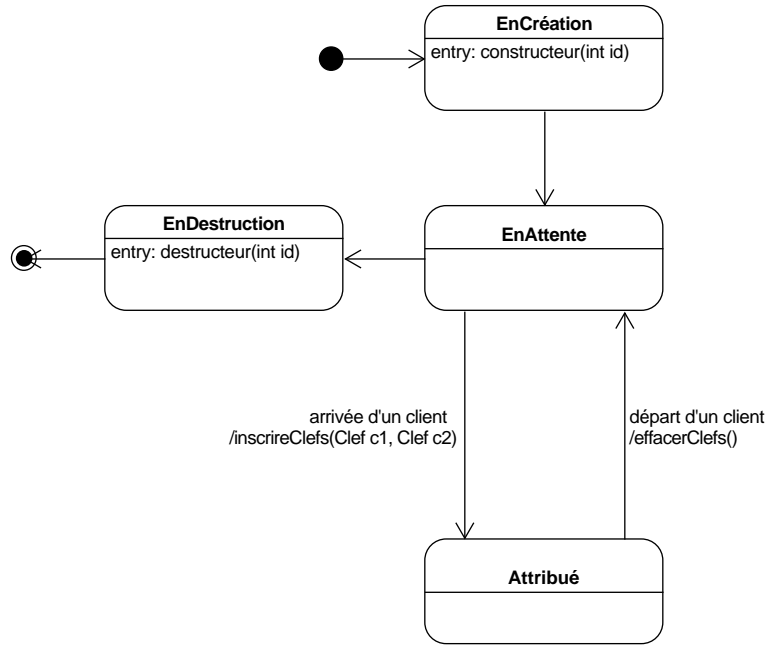


FIGURE 6 – Diagramme de machine à états de la classe « badge »

Invariant de la classe Badge : $(id \neq \text{null} \wedge \neg \text{vide}) \ \&\& \ (\neg \text{estAttribue} \wedge \text{premiereClef} \neq \text{null} \wedge \text{secondeClef} \neq \text{null}) \ || \ (\neg \text{nonAttribue} \wedge \text{premiereClef} = \text{null} \wedge \text{secondeClef} = \text{null})$

6 Préparation des tests unitaires

Numéro de test	1	2	3
Identifiant du badge bien formé \neq null $\wedge \neg$ vide	F	T	T
Badge inexistant avec cet identifiant		F	T
$/enAttente = true$			T
$/estAttribue = false$			T
invariant			T
Levée d'une exception	OUI	OUI	NON
Création du badge acceptée	F	F	T
Nombre de jeux de test	2	1	1

TABLE 7 – Table de décision des opérations pour Badge « constructeur »

Numéro de test	1	2
$/enAttente$	F	T
$/enAttente = false$		T
$/estAttribue = true$		T
invariant		T
Levée d'une exception	OUI	NON
Attribution du badge confirmée	F	T
Nombre de jeux de test	1	1

TABLE 8 – Table de décision des opérations pour Badge « estAttribué »