CSC4102 : Qualité du modèle UML et patrons de conception

Denis Conan, avec Paul Gibson

Janvier 2025

Sommaire

1 Motivations et objectifs

1.1 Qu'est-ce que la qualité d'un modèle UML ?
1.2 Pourquoi les patrons de conception ?
1.3 Quels sont les objectifs de la séance ?
1.4 Après la séance, à quoi ça sert ? où ? quand ?

1.1 Qu'est-ce que la qualité d'un modèle UML ?

- Quelques premiers éléments de réponse à partir de [Lange and Chaudron, 2004]
  « *An empirical assessment of completeness in UML designs.* »
  - *Well-formedness : The basic level of analysis is within a single diagram. In the semantics section of the UML specification well-formedness rules for model elements are defined.*
    - *These restrictions prevent designers from producing diagrams that contain serious flaws.*
  - *Consistency : Overlapping diagrams refer to common information. If two diagrams in the model contain contradictory common information, then there is an inconsistency between these diagrams.*
    - *If an inconsistency between two diagrams remains unresolved, miscommunication and high integration effort are likely to happen.*
  - *Completeness : A UML model is complete, if for each element in the one diagram it's expected counterpart in the other diagram is present.*
    - *Violations of model completeness can cause unimplemented functionality.*
      - Dans le module CSC4102, nous ne modélisons pas tout

1.2 Pourquoi les patrons de conception ?

Est-ce une conception réutilisable, évolutive, adaptable, performante...

Christopher Alexander,

A pattern language : towns, buildings, construction

[Alexander et al., 1977]

- *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*
- *A pattern is an instruction, which shows how this spatial configuration can be used, over and over again.*
- La différence entre les concepteurs novices et expérimentés est typiquement l'expérience
  - Le novice hésite beaucoup entre différentes variantes
  - L'expert trouve tout de suite la « bonne » solution (adaptable, évolutive...)

1.3 Quels sont les objectifs de la séance ?

- Vérifier la qualité du modèle de la solution actuellement proposée
  - Identification et correction des problèmes de cohérence :
    - Entre les diagrammes UML
    - Entre le modèle UML et le code JAVA
- Étudier l'application d'un patron de conception
  - Notre objectif n'est pas que vous deveniez des « experts », mais que vous connaissiez le concept pour avoir appliqué au moins un patron de conception

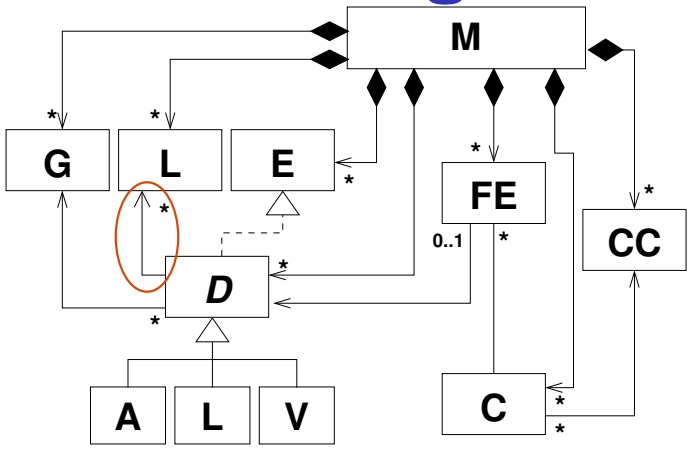
1.4 Après la séance, à quoi ça sert ? où ? quand ?

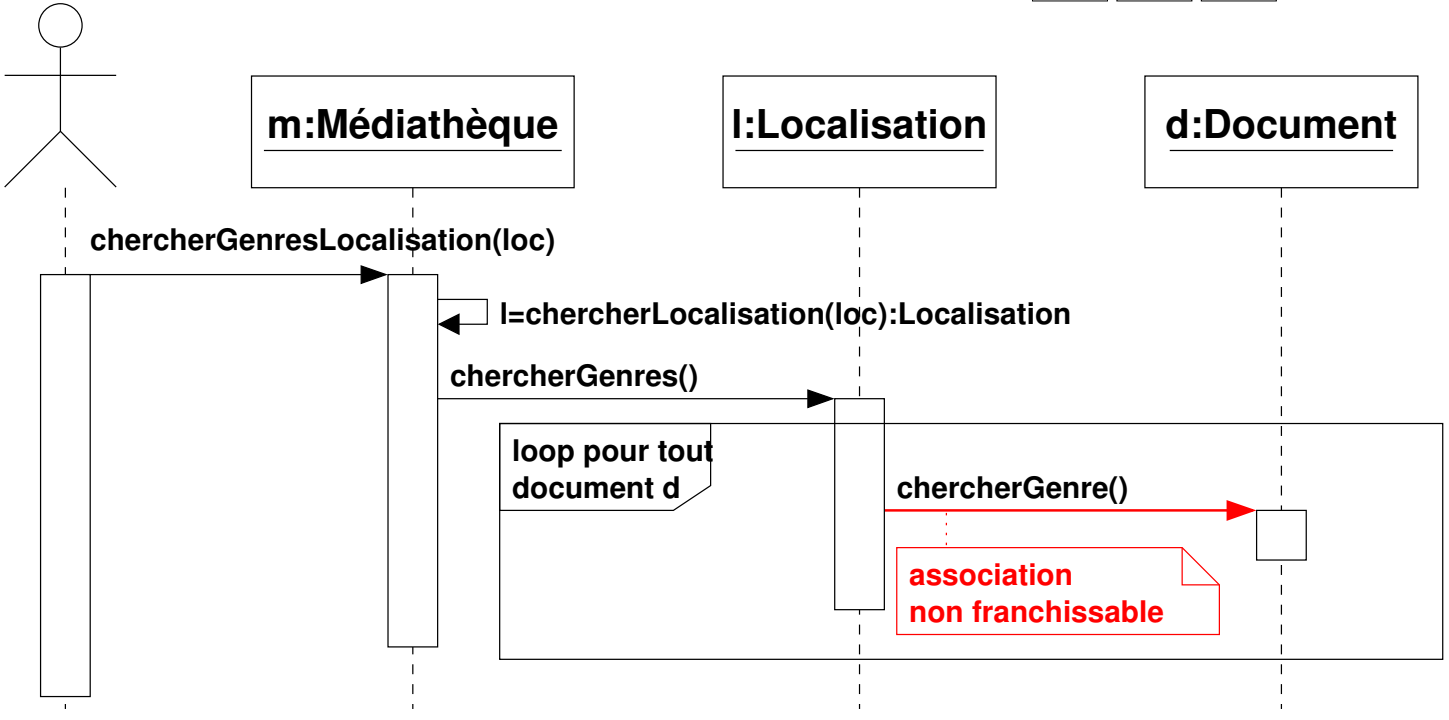
- En plus de la connaissance d'un domaine métier
  (par exemple le contrôle aérien, la gestion d'actifs, ou encore le vote électronique)
  la connaissance des patrons de conception est efficace
  - Partout : des sciences sociales au logiciel, en passant par la pédagogie
  - Nous sommes intrinsèquement bons dans la reconnaissance de patrons
- À l'avenir (après ce module), essayez l'approche de conception à base de patrons
  - À partir des exigences, considérez les concepts, les problèmes, les contextes, les attributs de qualité, et cherchez à appliquer les patrons de conception (génériques) avant de construire des solutions spécifiques
  - Littérature conseillée sur le sujet
    - Les patrons de conception du GoF (*Gang of Four*) [Gamma et al., 1994]
    - WikiLivre : « Patrons de conception » [Wikilivres, 2016]
    - *Java Design Patterns : Reusable Solutions to Common Problems* [Joshi, 2015]

2 Qualité du modèle UML

2.1 Cohérence : non-respect de la navigabilité

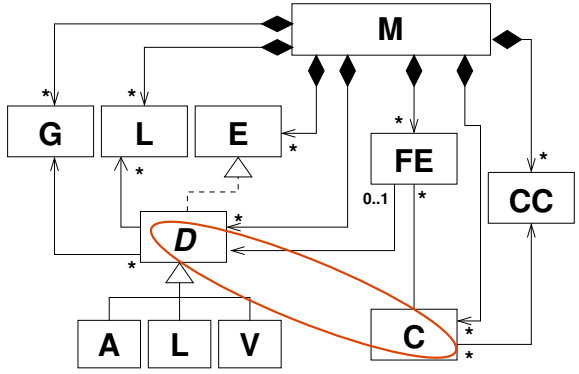
- Diagramme de séquence du cas d'utilisation « ajouter un document »
  - Selon le diagramme de classes, l'association `Localisation`—`Document` n'est pas franchissable vers `Document`

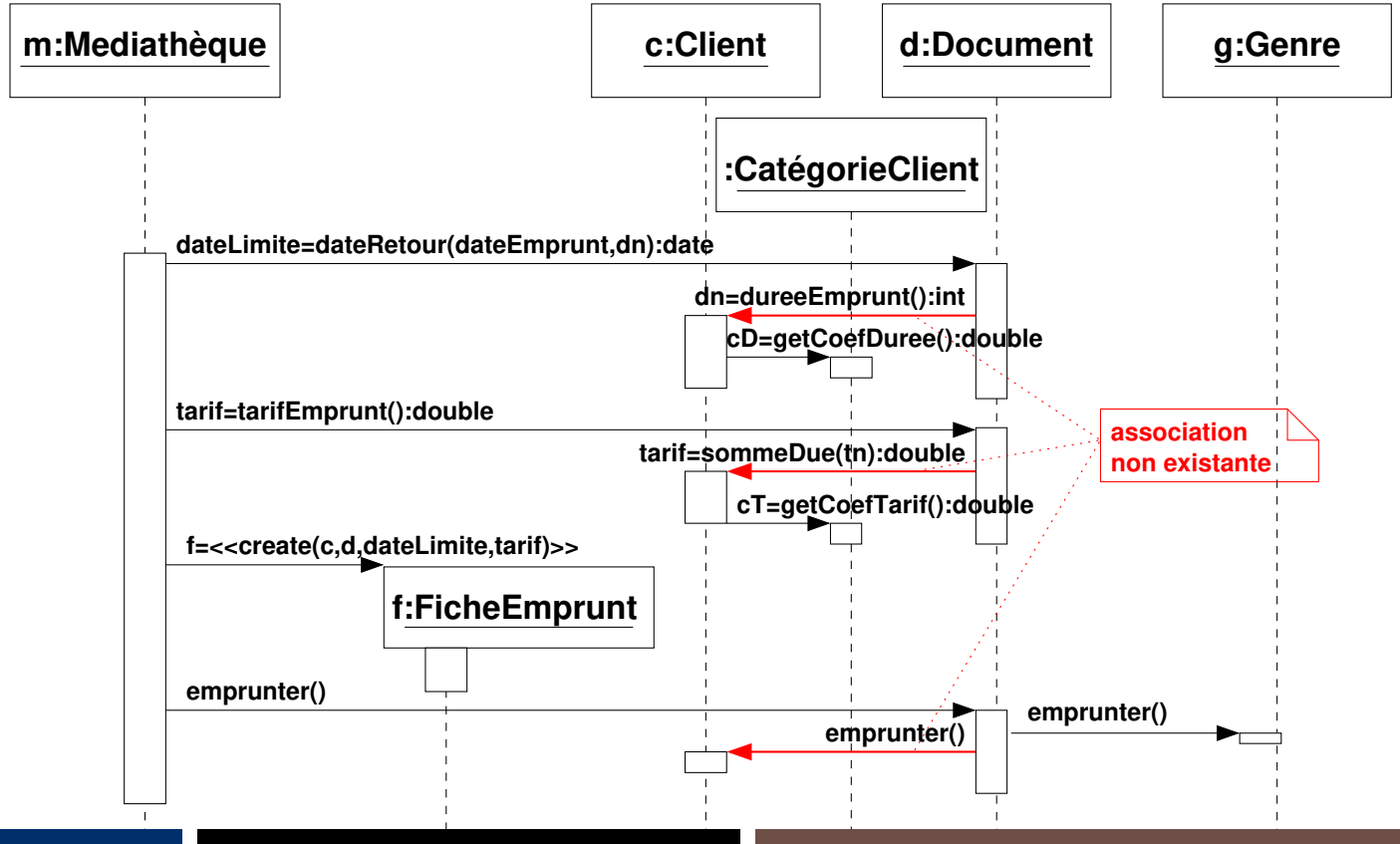




2.2 Cohérence : parcours d'association inexistante

- Diagramme de séquence du fragment « création de la fiche d'emprunt »
  - Dans le diagramme de classes, l'association `Document`—`Client` n'existe pas





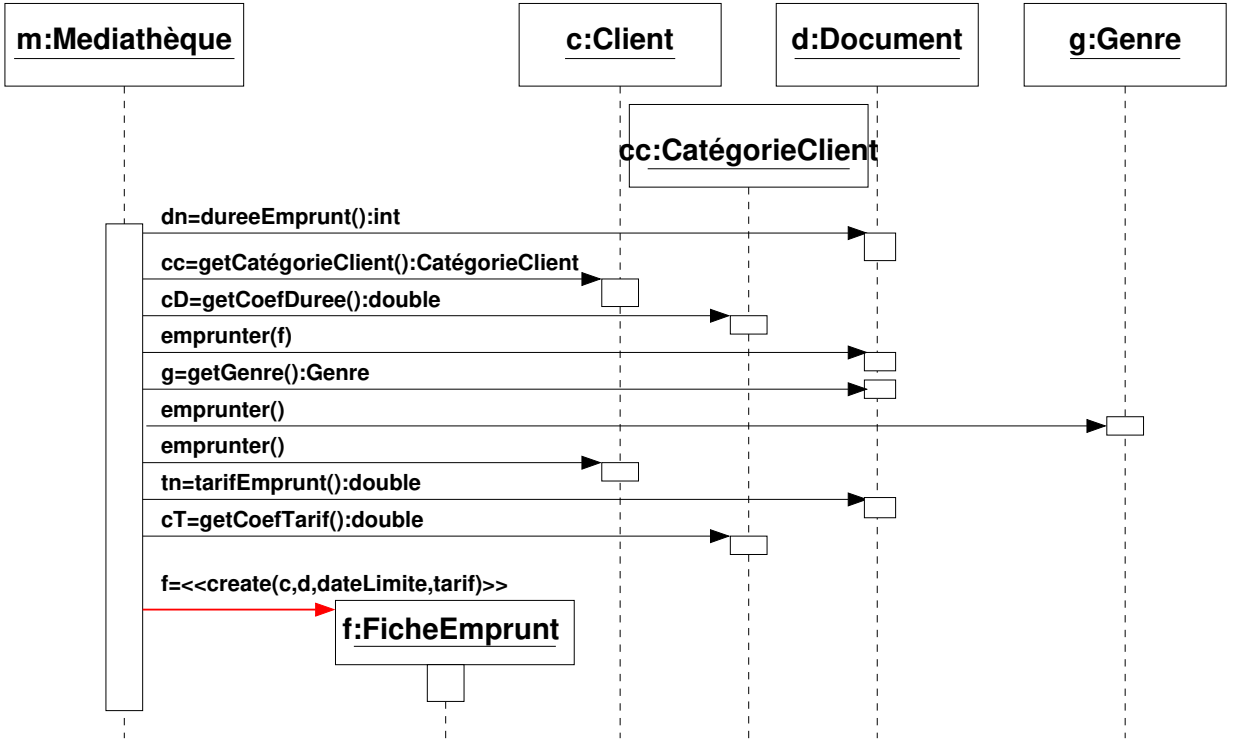
2.3 Principes généraux de la méthode Booch

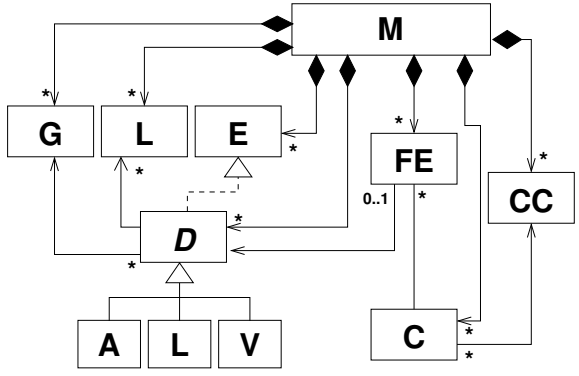
- *How can one know if a given class is well designed ?*

  « *Object Oriented Analysis and Design with Applications* » [Booch, 1991], pages 123–125

  - *Sufficient : enough characteristics of the abstraction to permit meaningful and efficient interaction*
  - *Complete : general enough to be commonly usable*
    - *Many high-level operations can be composed from low-level ones*
    - Attention cependant à l'encapsulation (p.ex. tous les *getters* ? non !)
  - *Primitive : [only] operations that can be efficiently implemented [with] the underlying representation of the abstraction*
  - *Coupling :*
    - *On the one hand, weakly coupled classes are desirable*
    - *On the other hand, inheritance —which tightly couples superclasses and their subclasses— helps us to exploit the commonality among abstractions*
  - *Cohesion :*
    - *The least desirable form of cohesion = coincidental cohesion : entirely unrelated abstractions are thrown into the same class*
    - *The most desirable form of cohesion = functional cohesion : the elements of a class all work together to provide some well-bounded behavior*

2.3.1 Exemple de questionnement de la cohésion





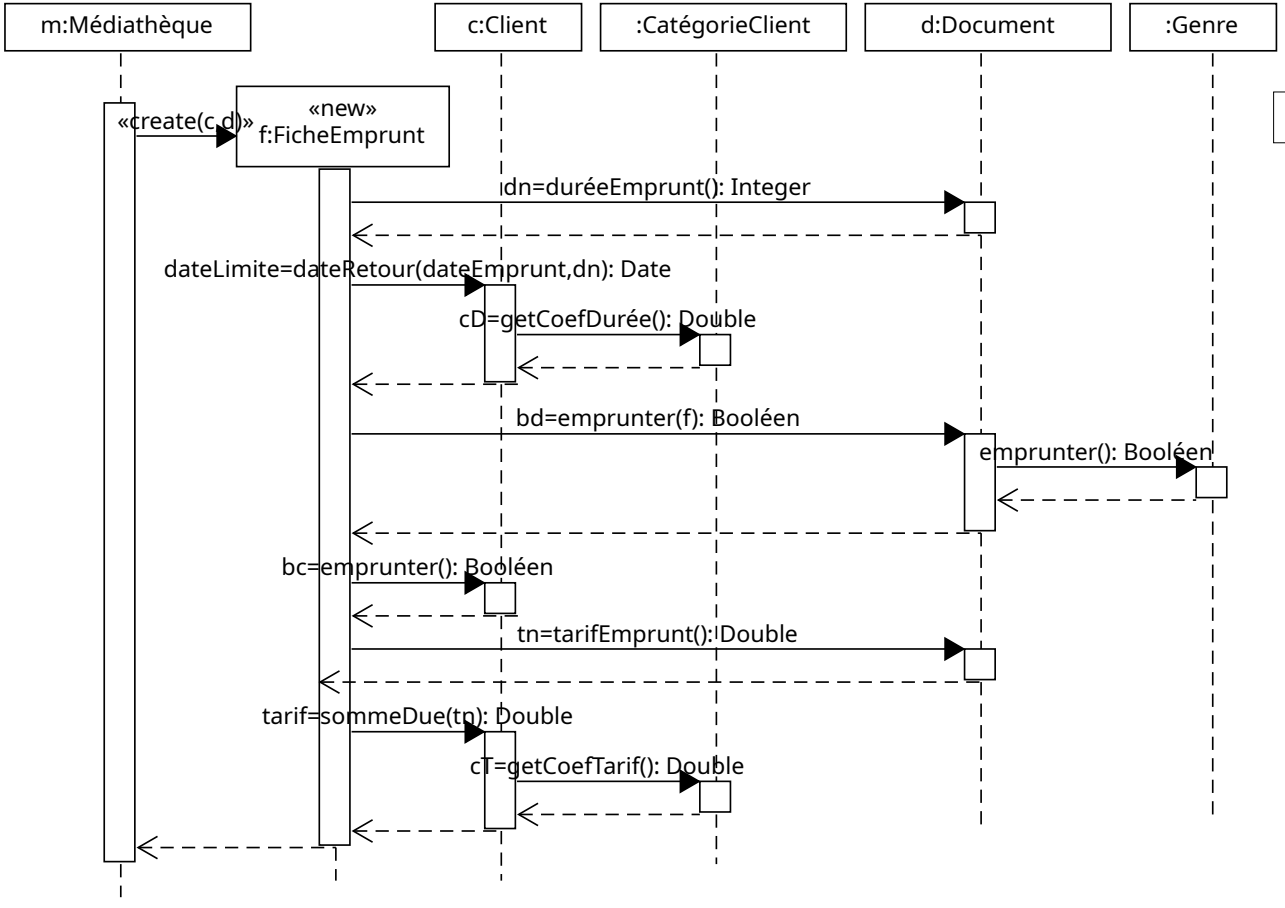
Cohésion fortuite : pourquoi toute l'algorithmique serait dans la façade ?

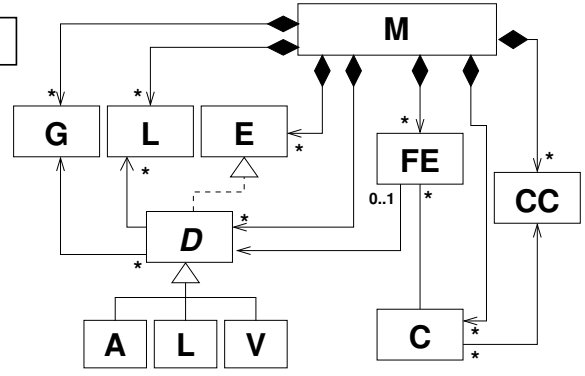
- ☹ Les autres classes se dégradent en simples structures de données (sans algo.)
  - ☞ Risque d'introduction de *getters*/*setters*, ce qui contredit l'ensapsulation

✓ Cohésion fonctionnelle : `FicheEmprunt` responsable de calculer `dateLimite`... ?

- ✓ Elle connaît via ses associations toutes les informations nécessaires

2.3.2 Exemple de modélisation préférée





- La règle de conception utilisée ici est plus précisément appelée la « loi de Déméter » [Lieberherr and Holland, 1989, Wikipedia, 2023] : « en particulier, un objet doit éviter d'invoquer des méthodes d'un membre objet retourné par une autre méthode »

3 Patrons de conception

3.1 Principe

E. Gamma, R. Helm, R. Johnson, J. Vlissides,
(le *Gang of Four, GoF*),
Design Patterns : Elements of Reusable Object-Oriented Software
[Gamma et al., 1994], ACM SIGSOFT Outstanding Research Award, 2010

- « Valeur de l'expérience », « sentiment de déjà vu »
  - « Si l'on parvenait à se souvenir des détails du problème antérieur, et de la façon dont il avait été résolu, on pourrait alors réutiliser cette expérience, au lieu d'avoir à la redécouvrir »
    - « Recueillir l'expertise en matière de conception orientée objet, sous forme de modèle de conception, aussi appelé patron de conception »
    - « Chaque patron de conception, systématiquement, nomme, explique et évalue un concept important et qui figure fréquemment dans les systèmes orientés objet »

3.2 Définition et éléments essentiels

Définition et éléments « du *GoF* » [Gamma et al., 1994]

- Définition :
  - C'est une description de classes et d'objets coopératifs que l'on a spécialisés pour résoudre un problème général de conception dans un contexte particulier
- Quatre éléments essentiels :
  - Nom : identification d'un concept, vocabulaire dans un catalogue
  - Problème : décrit la ou les situations dans lesquelles le patron s'applique
    - Expose le sujet à traiter et son contexte
    - Avec parfois une liste de conditions à satisfaire
  - Solution : décrit les éléments qui constituent la conception
    - Relations entre eux, leur part dans la solution, leur coopération
    - Description générique d'un problème de conception
  - Conséquences : effets résultants de l'application du modèle sur la conception ainsi que les variantes de compromis

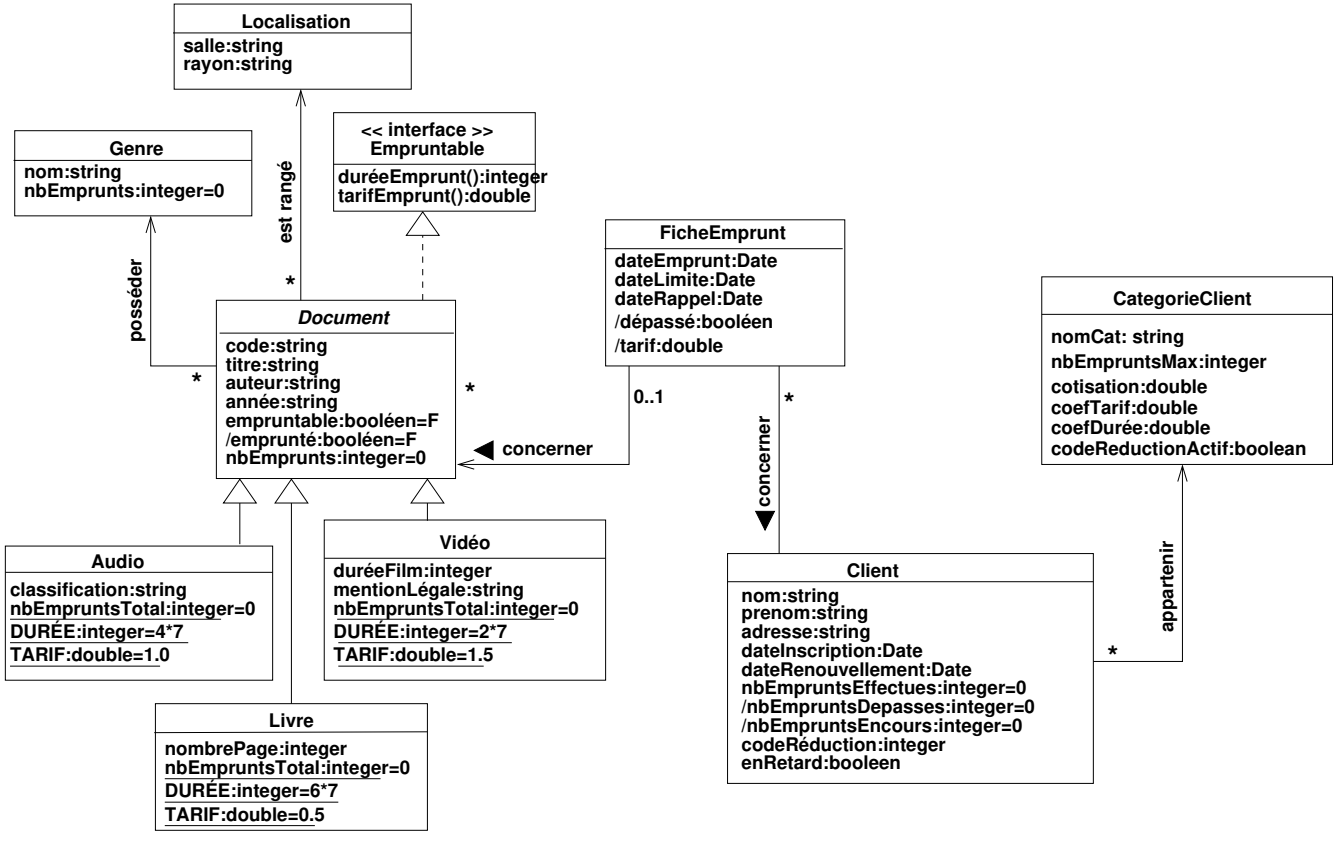
3.3 Catégories des patrons de conception « du *GoF* »

- Création : un patron de création permet de résoudre les problèmes liés à la création et la configuration d'objets
  - Singleton (*Singleton*), Fabrique abstraite (*Abstract Factory*), Constructeur (*Builder*), Fabrique (*Factory Method*), Prototype (*Prototype*)
- Structure : un patron de structure permet de résoudre les problèmes liés à la structuration des classes et leur interface en particulier
  - Façade (*Facade*), Adaptateur (*Adapter*), Pont (*Bridge*), Composite (*Composite*), Décorateur (*Decorator*), Poids-plume (*Flyweight*), Mandataire (*Proxy*)
- Comportement : un patron de comportement permet de résoudre les problèmes liés aux interactions entre les classes
  - Chaîne de responsabilité (Chain of responsibility), Commande (*Command*), Interpréteur (*Interpreter*), Itérateur (*Iterator*), Médiateur (*Mediator*), Memento (*Memento*), Publier—Souscrire (*Publish—Subscribe*), Observateur–Observable (*Observer*), État (*State*), Stratégie (*Strategy*), Patron de méthode (*Template Method*), Visiteur (*Visitor*)

3.4 Structure, patron de conception Façade I

1. Problème : dans l'étude de cas Médiathèque

   - Comment créer un nouveau client ?
     Dans une catégorie client, mais quid de la navigabilité ? et puis laquelle ?
     Commencer par créer la catégorie client, mais, comment ?



3.4 Structure, patron de conception Façade II

2. Solution :

   - Soit une méthode de classe dans la classe catégorie client
     - Cette méthode est un *Builder*
       Il faut connaître les noms des classes (c.-à-d. l'intérieur du système)
       - Il devient alors difficile de faire évoluer le système :
         - En cas de renommage, tous les logiciels des acteurs doivent évoluer
   - ✓ Soit un unique objet dédié, connu de l'extérieur, et à partir duquel les acteurs appellent les méthodes des cas d'utilisation
     - ✓ Cette nouvelle classe est la Façade

3. Conséquences :

   - Façade : tous les appels depuis l'extérieur passent par là

3.4.1 Explications complémentaires

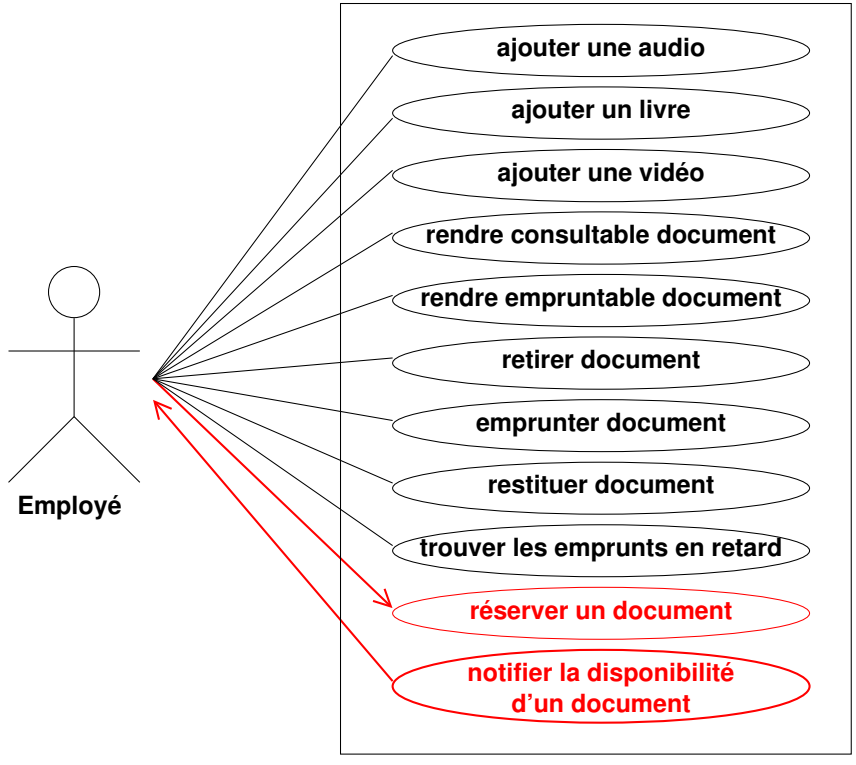
- La façade a pour but de cacher une conception
  - La façade peut être abstraite dans une interface
    - Une mise en œuvre peut être remplacée par une autre
- La façade permet de simplifier cette complexité en fournissant une interface simple du sous-système
- Habituellement, la façade est réalisée en réduisant les fonctionnalités tout en fournissant toutes les fonctions nécessaires aux utilisateurs
  - Par exemple, une façade peut être utilisée pour :
    - Rendre une bibliothèque plus facile à utiliser, comprendre et tester
    - Rendre une bibliothèque plus lisible
    - Réduire les dépendances entre les clients de la bibliothèque et le fonctionnement interne de celle-ci

3.5 Comportement, patron Publier-Souscrire I

1. Problème

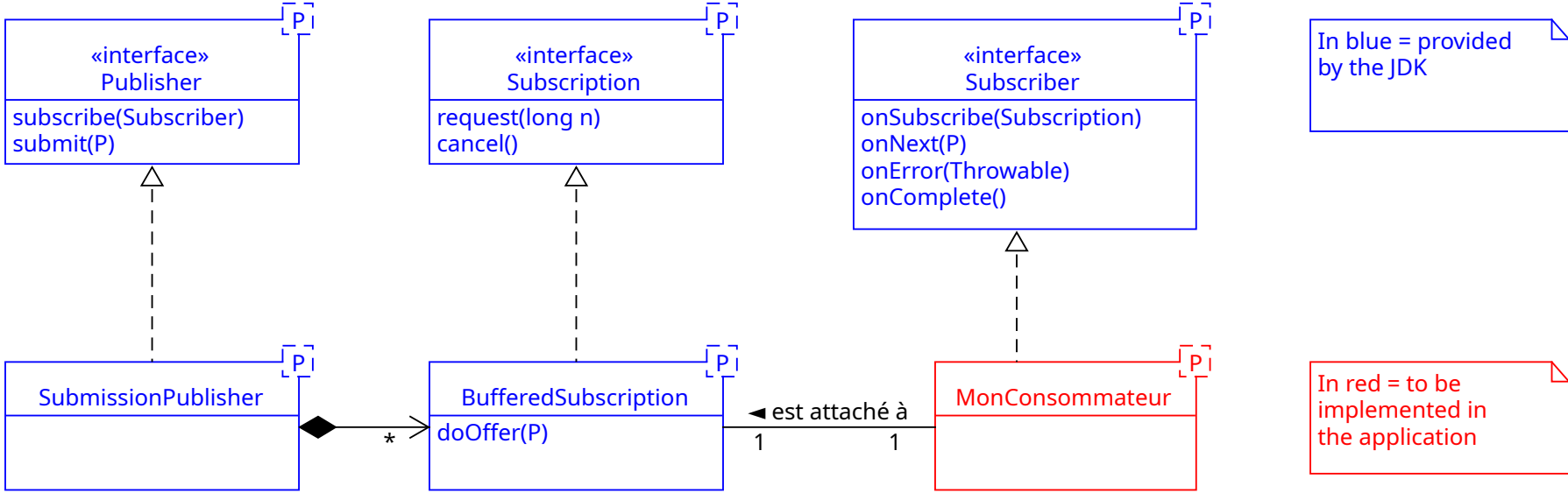
- Comment notifier un objet de manière asynchrone ?
  - Le producteur notifie le consommateur
  - Le message est dit asynchrone
  - Asynchronisme :
    production d'information $\Longrightarrow$
    « à terme » le consommateur est notifié ;
    le producteur n'attend pas que le consommateur soit notifié pour continuer son exécution
- Comment avoir plusieurs consommateurs enregistrés auprès d'un producteur ?

- P.ex., le système notifie l'acteur d'un événement



3.5 Comportement, patron Publier-Souscrire II

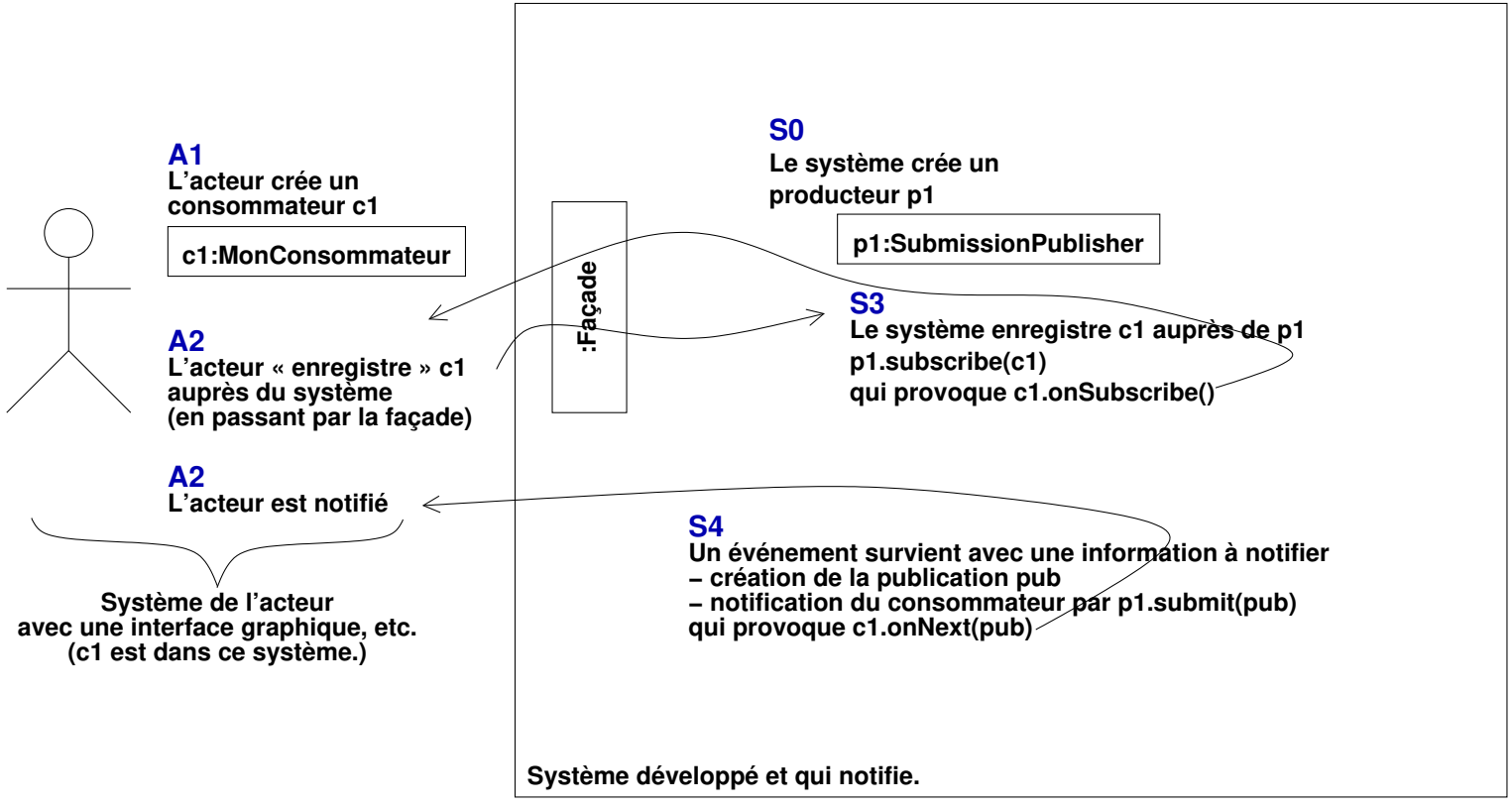
2. Solution (dans le contexte de la programmation en JAVA)



- `P` = type de la publication
- Producteur et consommateurs sont paramétrés avec `P`
- Le flux est géré dans `BufferedSubscription`, qui gère une collection de taille n de publications de type P

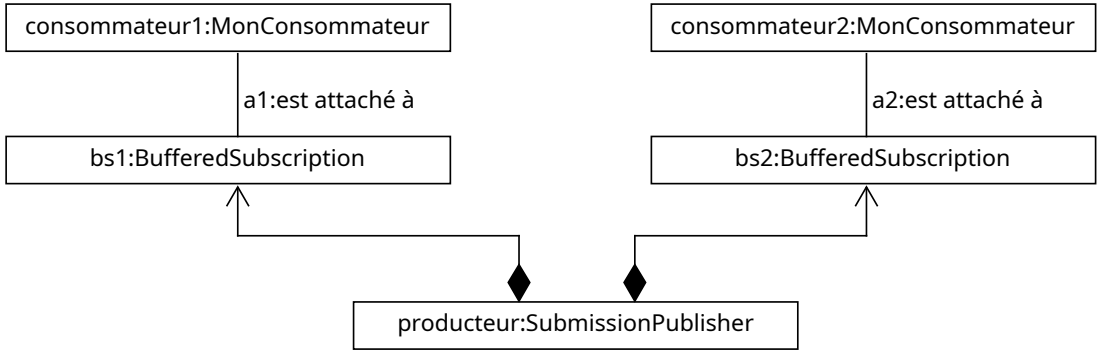
3.5 Comportement, patron Publier-Souscrire III

Visualisation des interactions avant de détailler le fonctionnement

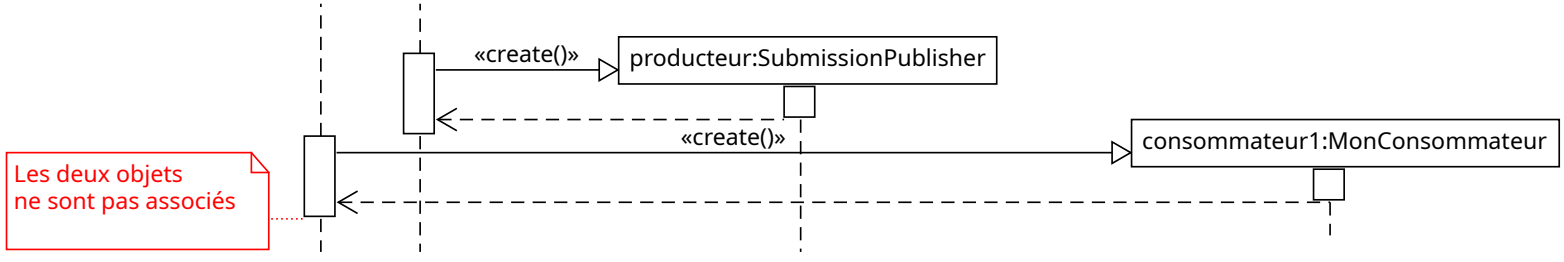


3.5 Comportement, patron Publier-Souscrire IV

Diagramme d'objets avec un producteur et deux consommateurs :

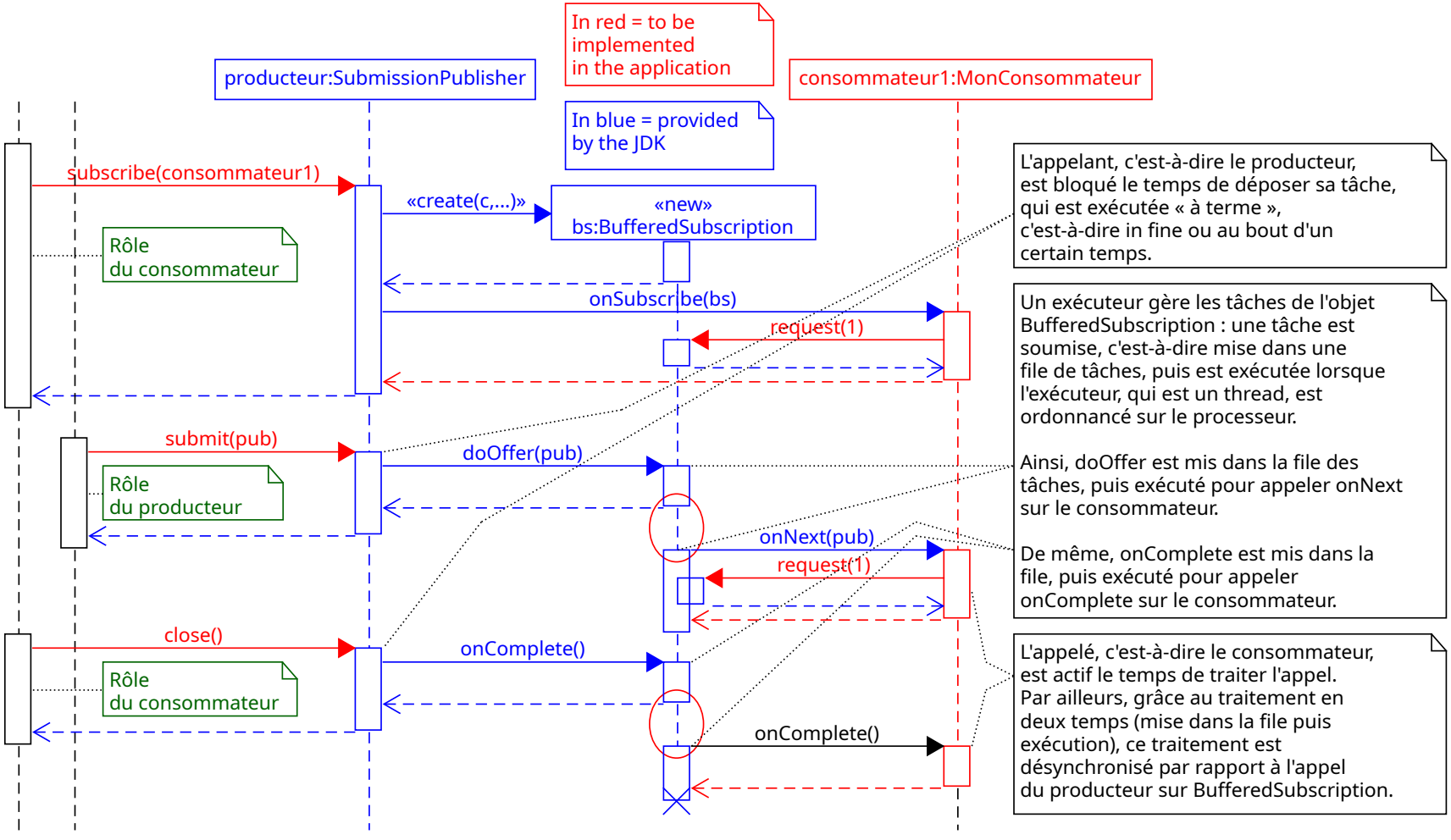


Début séquence pour la diapositive qui suit avec un seul consommateur :



3.5 Comportement, patron Publier-Souscrire V

Suite de la séquence : les objets `producteur` et `consommateur1` sont déjà créés



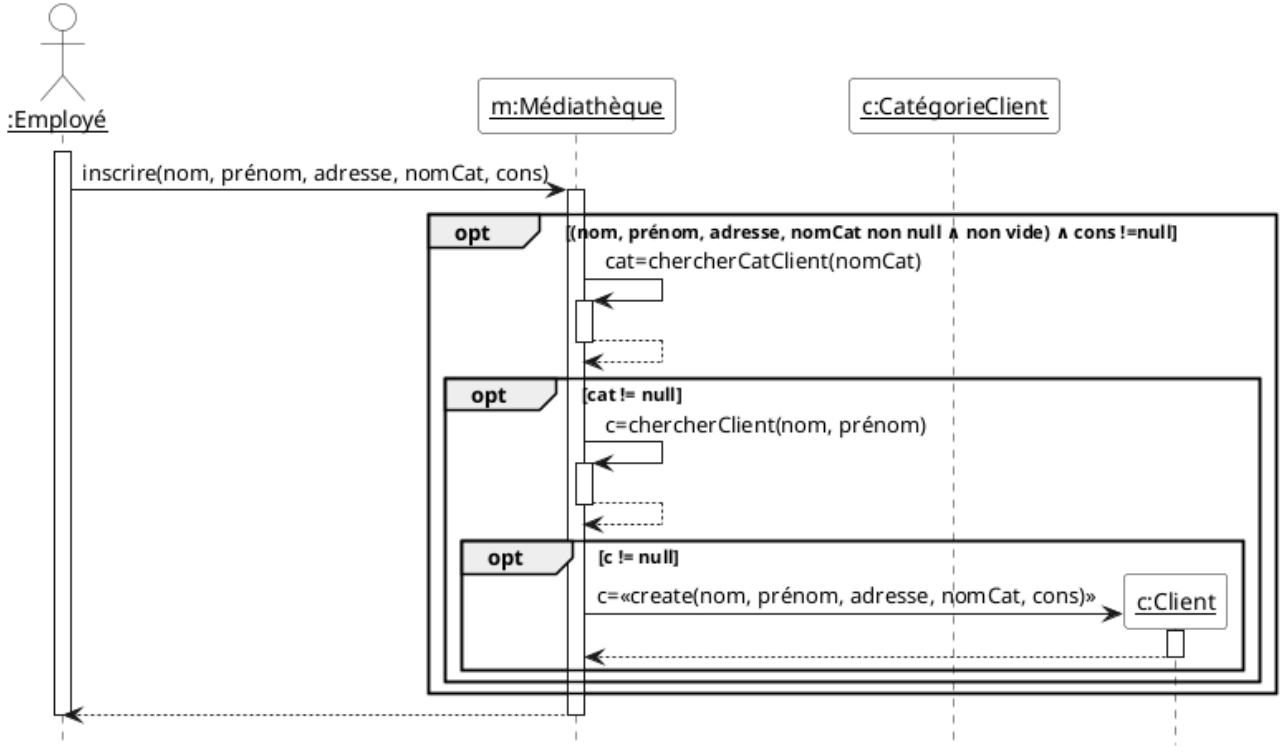
3.5.1 Exemple, insertion dans la médiathèque I

Insertion du patron de conception Publier/-Souscrire dans la modélisation de la médiathèque

- Dans le diagramme de classes
  - Classe `Client`, nouvel attribut `consoNotif`, qui est fourni par l'acteur `Client`
  - Classe `Document`, nouvel attribut `prodNotif` pour notifier de la disponibilité du document (lors de sa restitution)

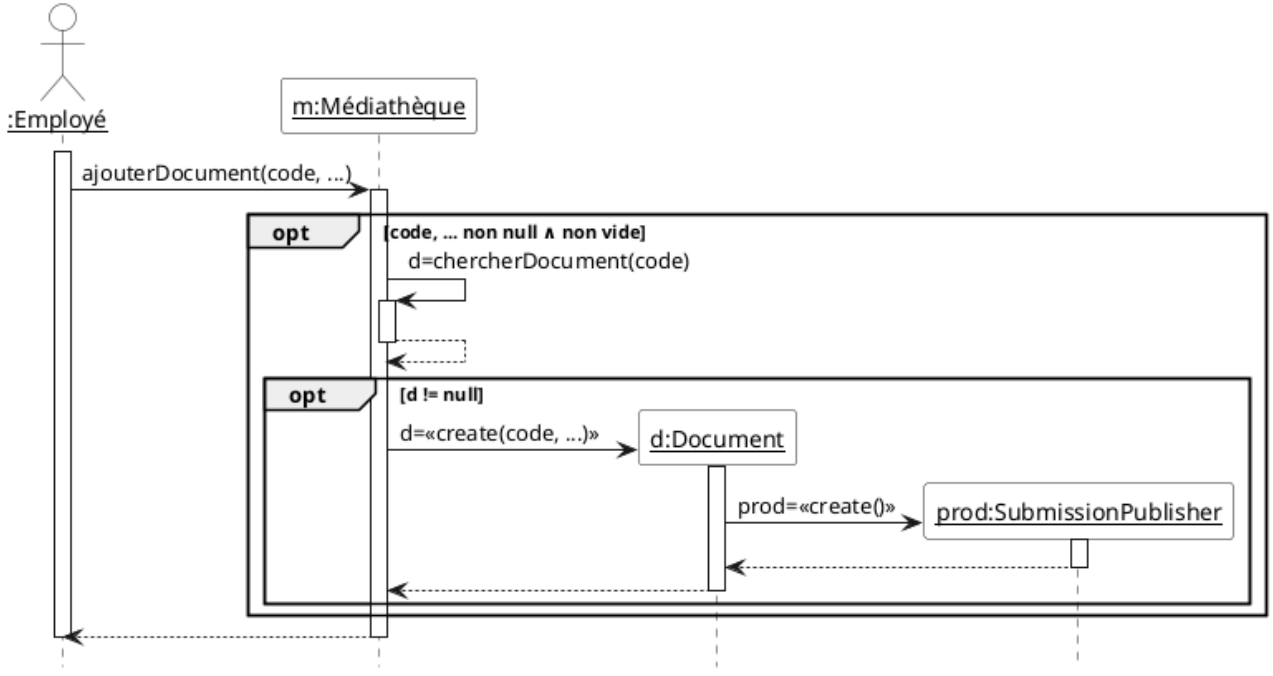
3.5.1 Exemple, insertion dans la médiathèque II

- Dans le diagramme de séquence « inscrire un client », enregistrement du consommateur
  - Simplification : on ignore les différences d'inscription fonction de la catégorie client



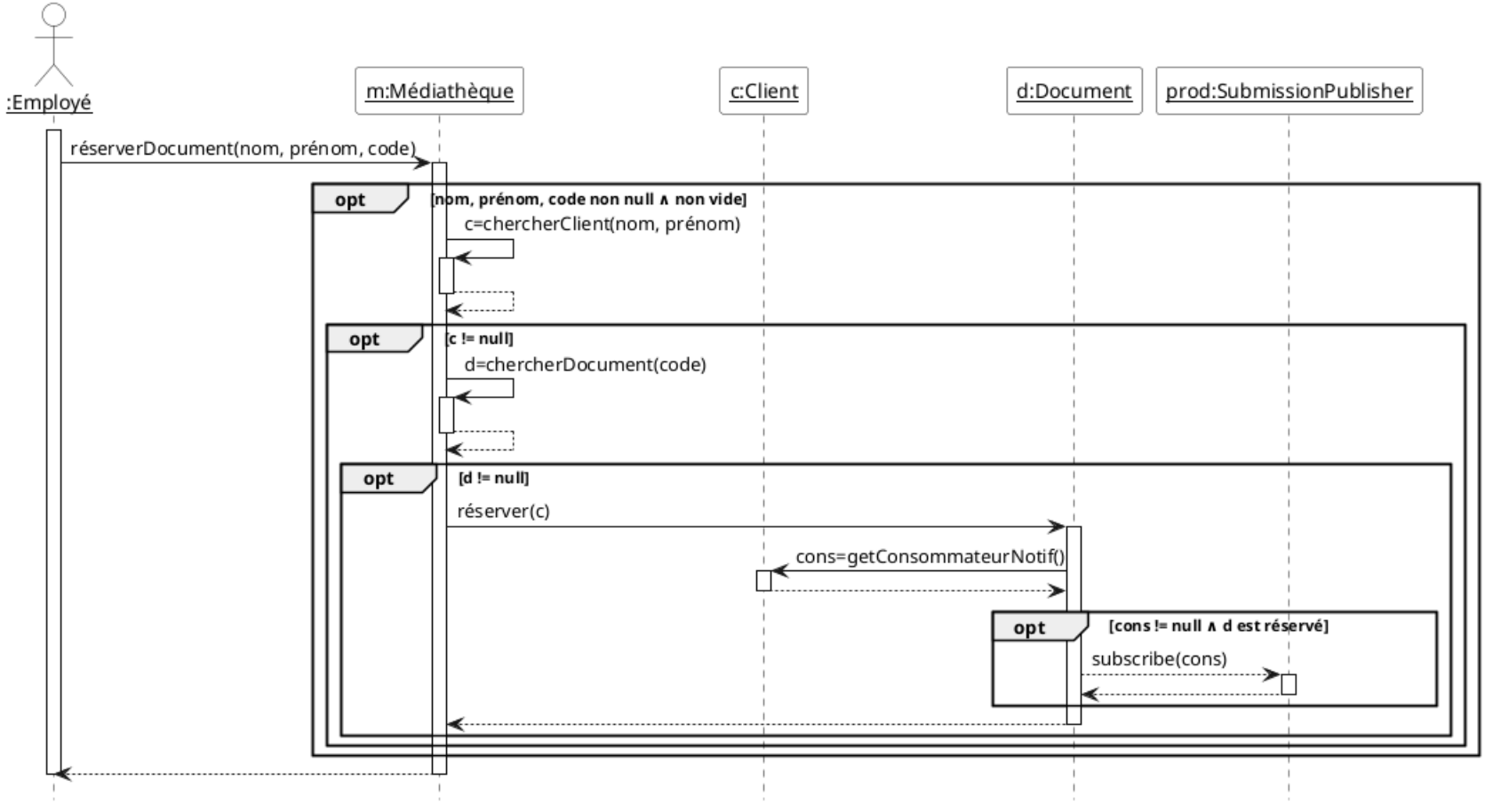
3.5.1 Exemple, insertion dans la médiathèque III

- Dans le diagramme de séquence « ajouter un document », création d'un producteur pour notifier les clients en attente d'emprunt
  - Simplification : on ignore les différents types de documents, le genre, etc.



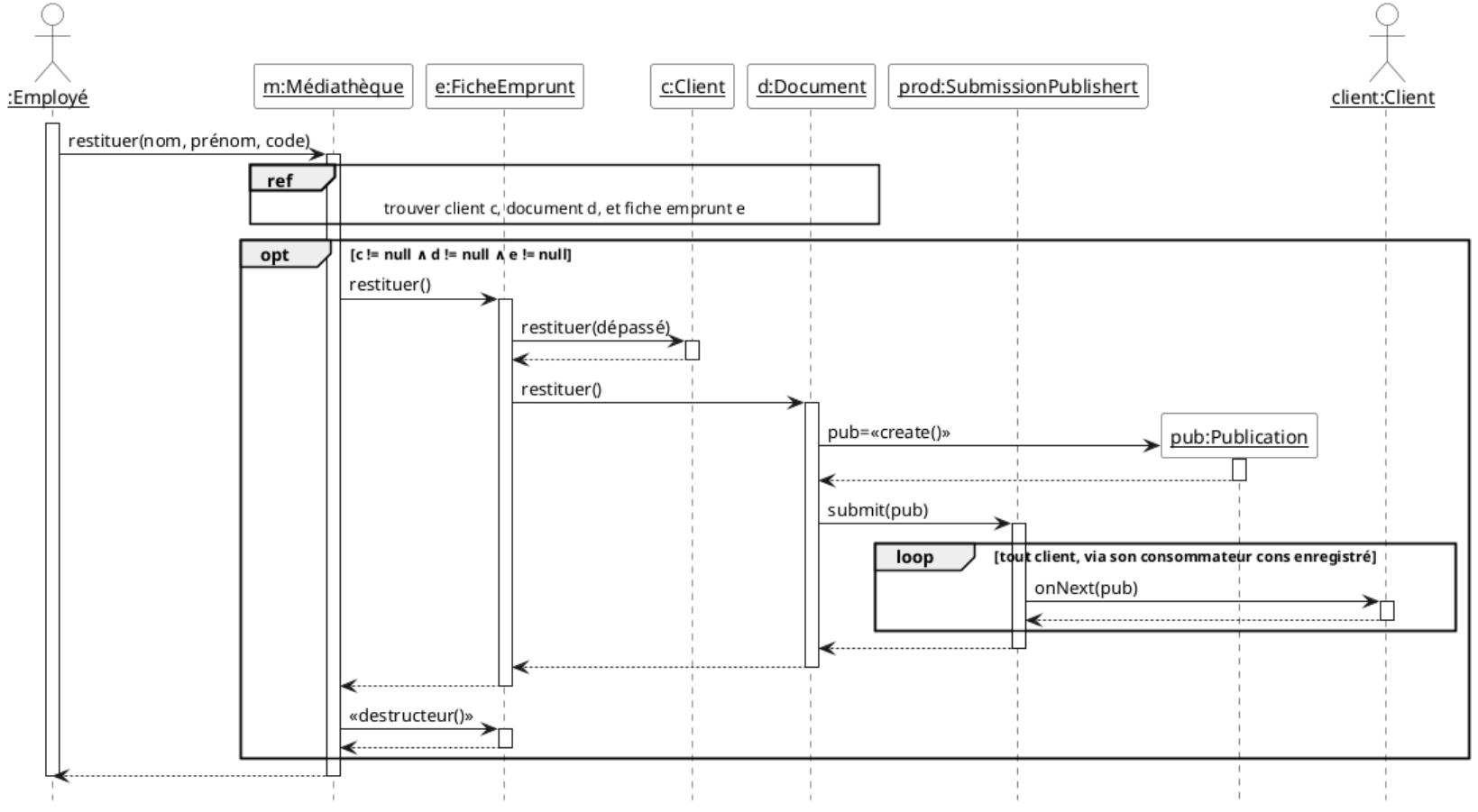
3.5.1 Exemple, insertion dans la médiathèque IV

- Dans le diagramme de séquence « réserver un document », souscription du consommateur du client au producteur du document d'emprunt



3.5.1 Exemple, insertion dans la médiathèque V

- Dans le diagramme de séquence « emprunter un document », notifier de la disponibilité les acteurs clients en attente : `submit(pub)` $\rightsquigarrow$ `onNext(pub)`



3.5.2 Exemple de code, consommateur

Classe seance9.patrondeconception.publiersouscrire.MonConsommateur

```
1  public class MonConsommateur implements Subscriber<Publication> {
2    private Subscription souscription;
3    public MonConsommateur(final String id) { // ...
4    @Override
5    public void onSubscribe(final Subscription souscription) {
6      this.souscription = souscription;
7      // on consomme un message des qu'il arrive ; un a la fois
8      // on declare qu'on est pret a recevoir un message
9      this.souscription.request(1);
10   }
11   @Override
12   public void onNext(final Publication publication) {
13     // reception d'une publication...
14     // on declare qu'on est pret a recevoir un nouveau message
15     souscription.request(1);
16   }
17   @Override
18   public void onError(final Throwable throwable) { //...
19     // erreur sur la gestion du flux, par exemple producteur.subscribe
20     // d'un consommateur qui est deja un subscriber du producteur
21     throwable.printStackTrace();
22   }
23   @Override
24   public void onComplete() { // lorsque le producteur ferme le flux
25 }
```

3.5.3 Exemple de code, scénario

Classe seance9.patrondeconception.publiersouscrire.Main

```
1  // creation du producteur
2  SubmissionPublisher<Publication> producteur = new SubmissionPublisher<>();
3  // creation de deux consommateurs
4  MonConsommateur consommateur1 = new MonConsommateur("1");
5  MonConsommateur consommateur2 = new MonConsommateur("2");
6  // les consommateurs sont enregistres aupres du producteur
7  producteur.subscribe(consommateur1);
8  producteur.subscribe(consommateur2);
9  // on attend un peu pour que tous les consommateurs soient enregsitres
10 Thread.sleep(100);
11 // production d'un message
12 producteur.submit(new Publication("publication␣1"));
13 // production d'un message
14 producteur.submit(new Publication("publication␣2"));
15 // il faut attendre un peu pour permettre la transmission du producteur
16 // vers les consommateurs
17 Thread.sleep(100);
18 producteur.close();
```

4 Mise en pratique en TP (2h) + HP (3h)

- Qualité de la conception
  - Cohérence entre les diagrammes et avec le code
  - Identification de faiblesses dans la conception, voire leur correction
- Intégration de patrons de conception pour améliorer et/ou faciliter la conception de certaines fonctionnalités
  - Cas d'utilisation notifiant un acteur
    en utilisant le patron de conception Publier—Souscrire
- Continuation du développement
- Rendu de la séance en HP (3h) : `PDF` + JAVA dans « `develop` »
- Compléments :
  - « Pour aller plus loin » sur les patrons de conception
  - Autres exemples : projet `csc4102-exemples`, documents `Modelisation/p*.pdf` et codes source dans les paquetages `designpattern/*`

Références I

Alexander, C., Ishikawa, S., and Silverstein, M. (1977).
A pattern language : towns, buildings, construction, volume 2.
Oxford University Press.

Booch, G. (1991).
Object-Oriented Analysis Design : With Application.
Benjamin-Cummings Publishing Co.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994).
Design Patterns : Elements of Reusable Object-Oriented Software.
Addison-Wesley.

Joshi, R. (2015).
Java Design Patterns : Reusable Solutions to Common Problems.
Exelixis Media P.C.

Lange, C. and Chaudron, M. (2004).
An empirical assessment of completeness in UML designs.
In *Proc. of the 8th International Conference on Empirical Assessment in Software Engineering*, pages 111–121.

Références II

Lieberherr, K. and Holland, I. (1989).
Assuring good style for object-oriented programs.
IEEE Software, 6(5) :38–48.

Wikilivres (2016).
Patron de conception.
Patronsdeconception.

Wikipedia (2023).
Loi de Déméter.
https://fr.wikipedia.org/wiki/Loi_de_DÃľmÃľter.