CSC4102 : Conception detaillée et préparation des tests unitaires

Denis Conan, avec Chantal Taconet, Christian Bac et Paul Gibson

Janvier 2025

Sommaire

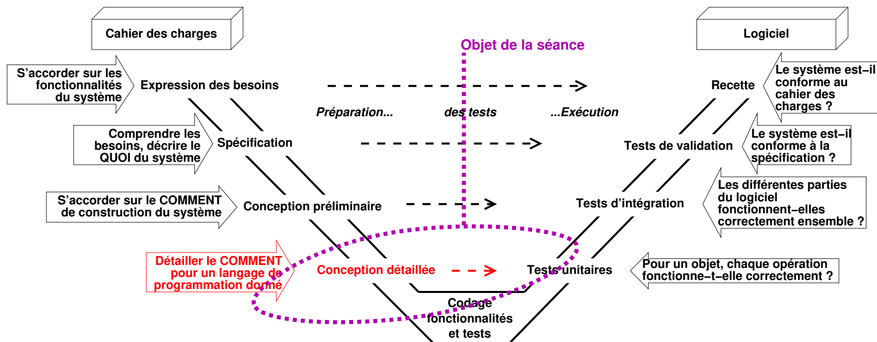
1. Motivations et objectifs de la séance
2. Conception détaillée
3. Préparation des tests unitaires
4. Mise en pratique en TP (2h) + HP (3h)

1 Motivations et objectifs de la séance

1.1 Contexte — Où en sommes-nous ?
1.2 Pourquoi la conception détaillée avant le codage ?
1.3 Pourquoi des tests unitaires ?
1.4 Quels sont les objectifs de la séance ?

1.1 Contexte — Où en sommes-nous ?

- Raffinement avec d'autres décisions à prendre, suite et fin de la conception
- Préparation des tests unitaires des opérations les plus importantes



1.2 Pourquoi la conception détaillée avant le codage ?

- Permettre une « ingénierie vers l'avant » (en anglais, *forward engineering*)
  - Raffinement, c.-à-d. ajout de contraintes pour simplifier la solution
    - P.ex., telle association, peut-on « la traverser » dans les deux sens ?
  - Modèle assez précis pour préparer des tests unitaires (voire générer du code)
- Prendre les dernières décisions indépendantes du langage de programmation
  - Si c'est une autre équipe qui code
    - Limiter les choix et les interprétations en rassemblant dans une fiche tout ce qui concerne une classe
      - P.ex., tel attribut dérivé, est-ce une information mise à jour ou calculée à la demande ?
- Choisir des canevas logiciels (bibliothèques) techniques[1]

1. Ces aspects ne sont pas abordés dans ce module, mais lors des VAP comme ASR, DSI ou JIN

1.3 Pourquoi des tests unitaires ?

Enquête sur le développement logiciel dans les *startups* [Giardino et al., 2016]

- *The highest priority = a functioning but faulty product to quickly market*
- *Startups will likely increase their user-base, product size, and number of developers.*
  This will require [...] to eventually pay the accumulated technical debt.

- *Technical debt [Fowler, 2003, Cunningham, 2009]*
  - *Developers accept compromises in a system in one aspect (e.g. testing) to meet an urgent demand in some other aspects (e.g. a deadline) ; such compromises incur a "debt" on which "interest" has to be paid*
  - *Some of the five dimensions of technical debts : design, testing, code*
- Pour évaluer la dette, il faut avoir un jour pratiqué « idéalement », ou proche de l'idéal : conception avec assez de détails pour préparer des tests unitaires = Rôle de ce module, et plus particulièrement de cette séance

1.4 Quels sont les objectifs de la séance ?

- Conception détaillée
  - Raffinement (ajout de contraintes) du diagramme de classes pour faciliter la programmation
  - Modélisation du cycle de vie des objets des classes principales dans des diagrammes de machine à états pour en déduire leur invariant
  - Le détail des classes dans une fiche par classe
    - Tous les attributs entièrement définis (p.ex. ceux traduisant des associations)
    - Liste de toutes les opérations (p.ex. trouvées dans les diag. de séquence)
- Préparation des tests unitaires sur les classes principales
  - Construire les tables de décision des opérations principales de ces classes
    - D'abord, pour les constructeurs
    - Puis, pour les opérations qui font changer l'état de l'objet

2 Conception détaillée

2.1 Raffinement du diagramme de classes pour ajouter des contraintes
2.2 Diagramme de machine à états
2.3 2è raffinement : traduction pour préparer la programmation

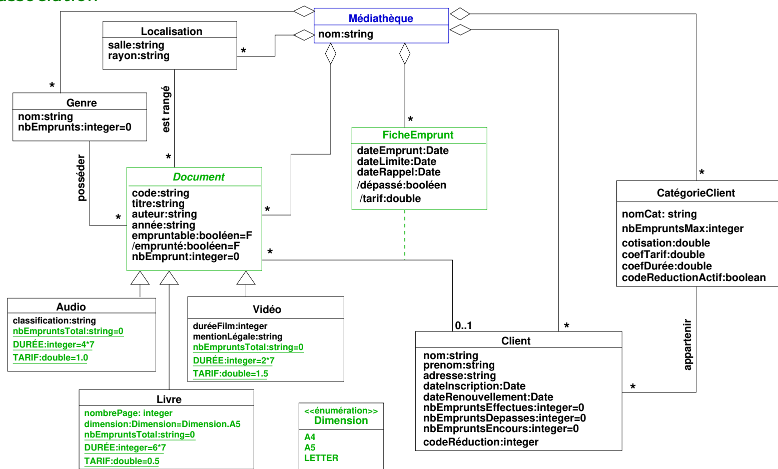
2.1 Raffinement du diagramme de classes pour ajouter des contraintes

2.1.1 Rappel, diagramme de classes préliminaire
2.1.2 Navigabilité
2.1.3 Composition = agrégation forte
2.1.4 Rappel, encapsulation et visibilité
2.1.5 Traduction des attributs dérivés

. Dans cette seconde étude du diagramme de classes, sont laissés pour la programmation les éléments de modélisation suivants : interface et classe paramétrée

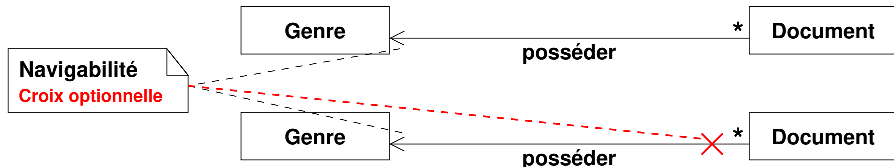
2.1.1 Rappel, diagramme de classes préliminaire

Classe, attribut (dérivé), opération, association, multiplicité, agrégation, généralisation spécialisation, agrégation, attribut/opération de classe, classe abstraite, classe d'association



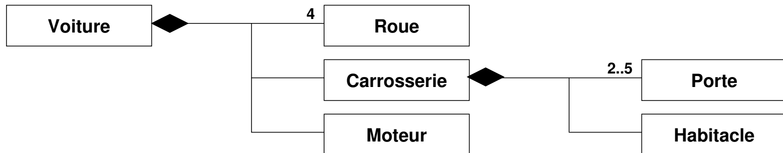
2.1.2 Navigabilité

- Par défaut, une association est bidirectionnelle
- Il est possible de la rendre unidirectionnelle
  (« on ne peut par aller contre la flèche »)
  - Il en résulte une simplification de la solution
    - Pas besoin du maintien de la cohérence des références croisées
      - « Ajouter un document $\Longrightarrow$ ajouter la connaissance du genre par le doc. »
        est plus facile à gérer que
      - « Ajouter un document $\Longrightarrow$ ajouter la connaissance du genre par le doc.
        + ajouter la connaissance du document par le genre »



2.1.3 Composition = agrégation forte

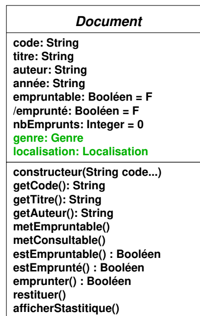
- Composition = agrégation forte liant les cycles de vie du composé (ensemble) et des composants (éléments) [OMG, 2013]
  - *A part object is included in at most one composite object at a time*
  - *All of the part instances that are objects are deleted with it*
  - *A part object may be removed before the composite object*
  - *The order and way in which composed objects are created is intentionally not defined*

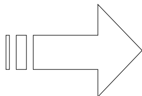


P.ex., voiture chez un concessionnaire *versus* dans une « casse » (en pièces détachées)

2.1.4 Rappel, encapsulation et visibilité

- Doit-on accéder à tous les attributs et à toutes les opérations d'un objet ? Non, c'est le principe de l'encapsulation
  - Visibilité `privé`, notation UML « − » : uniquement l'intérieur de la classe
  - Visibilité `protégé`, notation UML « # » : `privé` + les classes enfants
  - Visibilité `public`, notation UML « + » : toutes les classes



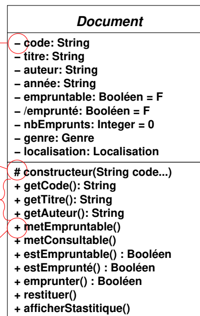


Par défaut, visibilité « privé » pour les attributs

Classe abstraite => constructeur uniquement pour les classes enfants

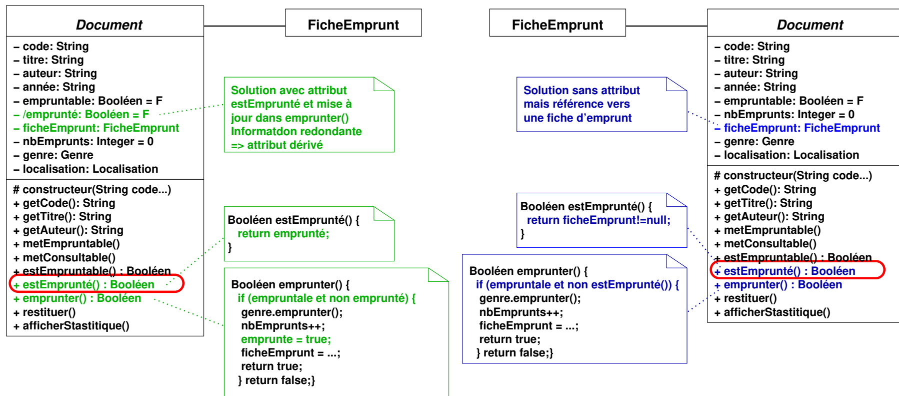
Attributs privés => si besoin, ajoutez des opérations get et/ou set

La plupart du temps, visibilité « public » pour les opérations



2.1.5 Traduction des attributs dérivés

- Opération de (re-)calcul ou de (simple) lecture
  selon que l'attribut est ou non recalculé à chaque fois que sa valeur est lue



2.2 Diagramme de machine à états

2.2.1 Pourquoi modéliser les états d'un objet ?
2.2.2 Types d'états, événement et transition
2.2.3 Transition : événement, condition, action
2.2.4 Transition implicite et actions liées à un état

. Les éléments suivants de la spécification UML ne sont pas étudiées : état composite, région, sous-machine, et pseudo-état

2.2.1 Pourquoi modéliser les états d'un objet ?

- Certaines classes sont plus complexes, et surtout,
  influent de manière importante sur le comportement global du système
- Il est important de décrire les événements provoquant les changements d'états des objets de ces classes
- Les changements d'états sont décrits dans des machines à états

- P.ex. dans la médiathèque, questions pour lesquelles les diagrammes de machine à états sont une aide :
  - Un document est-il empruntable ? est-il emprunté ?
  - Un emprunt est-il « en retard » ?
  - Un client peut-il emprunter ?

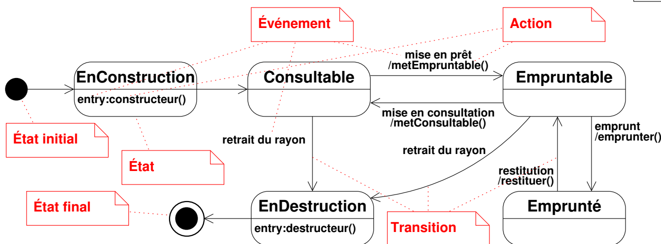
$\Longrightarrow$ Diagramme de machine à états pour `Document`, `FicheEmprunt` et `Client`

2.2.2 Types d'états, événement et transition

Document

code:string
titre:string
auteur:string
année:string
empruntable:booléen=F
/emprunté:booléen=F
nbEmprunts:integer=0

- Regardez les attributs significatifs, p.ex. les booléens



- CSC4102 : nous mettons toujours les états `EnConstruction`, `EnDestruction`
- En plus des états initial, final, `EnConstruction` et `EnDestruction`, ne conservez que les états significatifs, c.-à-d. dans lesquels l'objet reste pendant un certain temps et qui influent sur le comportement du système

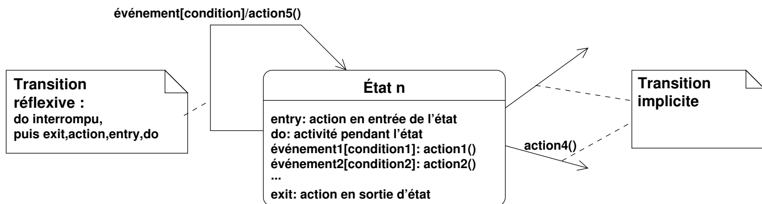
2.2.3 Transition : événement, condition, action

- Syntaxe complète d'une transition : `événement[condition]/action`
  - La transition est déclenchée par un événement
  - La transition est effectivement franchie si la condition est valide
    - Condition (uniquement) sur l'état de l'objet
  - Le franchissement déclenche l'exécution de l'action de la transition
    - L'exécution de l'action est considérée comme atomique[2]
    - L'action est une opération de la classe de l'objet
- P.ex. « résiliation du client [pas d'emprunt] / résilier() »
- Si plusieurs transitions sont franchissables, alors la transition effectivement franchie est choisie aléatoirement

2. Qui s'exécute comme un tout, sans pouvoir être interrompue ; on modélise l'état du système avant ou après l'action, pas au milieu.

2.2.4 Transition implicite et actions liées à un état

- Transition implicite $= \nexists$ événement $\wedge \nexists$ condition
  Donc, le franchissement est possible
- Actions exécutées à l'entrée et à la sortie d'un état (`entry` et `exit`)
- Action exécutée pendant toute la durée de présence dans l'état (`do`)
- Action interne déclenchée par un événement (p.ex. `événement1`)

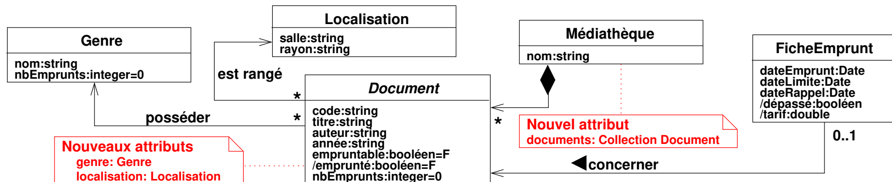


2.3 2è raffinement : traduction pour préparer la programmation

2.3.1 Traduction des associations en attributs
2.3.2 Traduction des agrégations/compositions
2.3.3 Rappels, « façade » et classe d'association
2.3.4 Traduction des diagrammes de séquence en algorithmes
2.3.5 Traduction des diagrammes de machine à états
2.3.6 Invariant de classe
2.3.7 Précondition, postcondition, et algorithme d'une opération I

2.3.1 Traduction des associations en attributs

- Référence = À partir d'un Document, « aller vers » un objet Genre

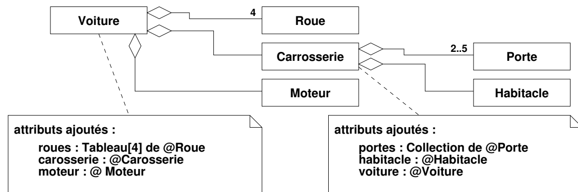


- Association unidirectionnelle = pas d'attribut du « côté de la flèche »
- Association binaire = 1 attribut ; association n-aire = $n - 1$ attributs
- Nom de l'attribut = nom du rôle ou forme nominale du nom de l'association
- Multiplicité « 1 » traduit par « `Classe` »
- Multiplicités « $0..N$ » et « $*$ » traduit par « `Collection Classe` »[3]

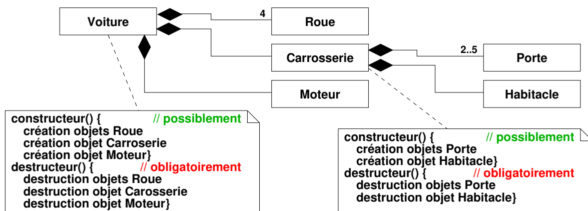
3. Choix du type de collection par le programmeur : liste [ordonnée], dictionnaire [trié]

2.3.2 Traduction des agrégations/compositions

- Agrégation = association, avec les mêmes règles de traduction en attributs

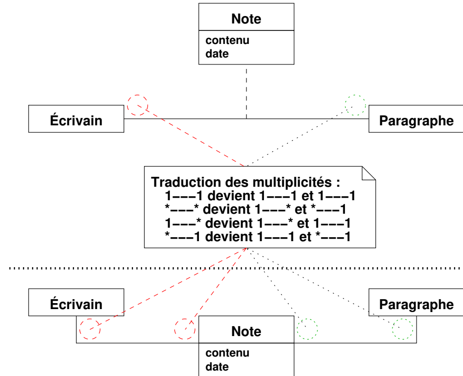


- Composition = ajouter le contrôle du cycle de vie des composants par le composé

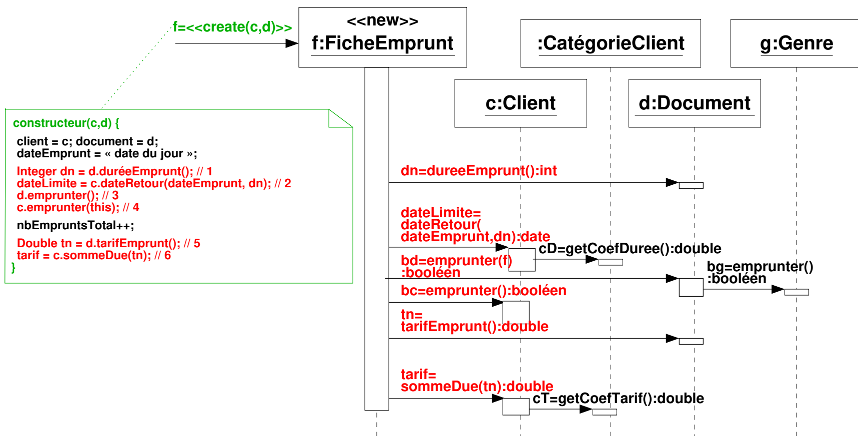


2.3.3 Rappels, « façade » et classe d'association

- Façade :
  - Constructeur et destructeur : cf. traduction des agrégation et composition
  - Une opération publique par cas d'utilisation
- Classe d'association :
  - Concept n'existant pas en langage de programmation
    $\Longrightarrow$ Traduction en deux associations lors de la conception détaillée



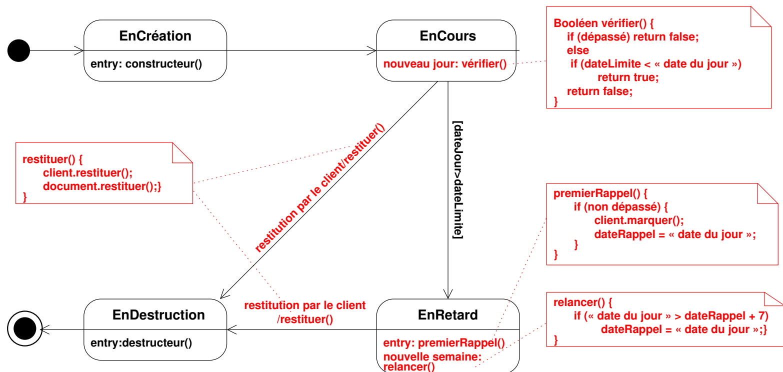
2.3.4 Traduction des diagrammes de séquence en algorithmes



. Ce diagramme de séquence est une partie de celui du cas d'utilisation « emprunter ».

2.3.5 Traduction des diagrammes de machine à états

- Certaines méthodes apparaissent en tant qu'action dans les diagrammes de machine à états



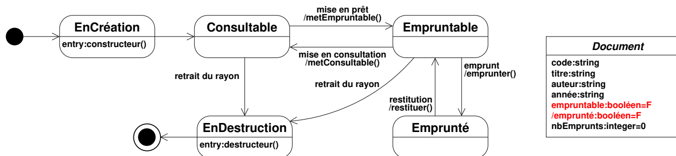
2.3.6 Invariant de classe I

- Invariant = Propriété vraie pendant tout le cycle de vie [Liskov and Wing, 1994, Lamport, 2003]
  - Ici, c'est le cycle de vie des instances de la classe
  - Exprimé en fonction des attributs modifiables
- Principe : étant donné un objet correctement construit, l'invariant reste vrai
- Méthodologie :
  - Calculez l'invariant
    - À partir du diagramme de machine à états et de la liste des attributs modifiables
    - Avec les contraintes de valeurs des attributs non modifiables
  - Ajoutez l'opération publique « `invariant(): booléen` »
  - Utilisez cette opération pour vérifier le bon fonctionnement des opérations qui modifient l'état

2.3.6 Invariant de classe II

P.ex., invariant de la classe `Document`

- Première étape : caractériser les états autorisés et ceux non autorisés



$(\neg \mathrm{empruntable} \wedge \neg \mathrm{emprunte}) \vee (\mathrm{empruntable} \wedge \neg \mathrm{emprunte}) \vee (\mathrm{empruntable} \wedge \mathrm{emprunte})$

$\equiv \langle$ distributivité de $\vee$ sur $\wedge \rangle$

$\left((\neg \mathrm{empruntable} \vee \mathrm{empruntable}) \wedge \neg \mathrm{emprunte}\right) \vee (\mathrm{empruntable} \wedge \mathrm{emprunte})$

$\equiv \langle$ principe du tiers exclu $(p \vee \neg p = true)$ et identité de $\wedge$ $(p \wedge true = p)\rangle$

$\neg \mathrm{emprunte} \vee (\mathrm{empruntable} \wedge \mathrm{emprunte})$

$\equiv \langle$ distributivité de $\vee$ sur $\wedge$, principe du tiers exclu et identité de $\wedge \rangle$

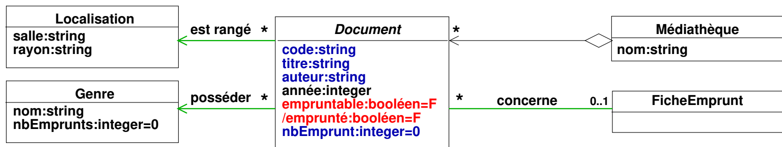
$(\neg \mathrm{emprunte} \vee \mathrm{empruntable})$

$\equiv \langle$ définition de $\implies \rangle$

$\mathrm{emprunte} \implies \mathrm{empruntable}$

2.3.6 Invariant de classe III

- Seconde étape : prise en compte des autres attributs



$\wedge$ (emprunte $\implies$ empruntable)
$\wedge$ code $\neq$ null $\wedge$ code $\neq$ vide
$\wedge$ titre $\neq$ null $\wedge$ titre $\neq$ vide
$\wedge$ auteur $\neq$ null $\wedge$ auteur $\neq$ vide
$\wedge$ nbEmprunts $\geq$ 0
$\wedge$ localisation $\neq$ null
$\wedge$ genre $\neq$ null
$\wedge$ $\left[(\neg \text{emprunte} \wedge \text{ficheEmprunt} = \text{null}) \vee (\text{emprunte} \wedge \text{ficheEmprunt} \neq \text{null})\right]$

2.3.7 Précondition, postcondition, et algorithme d'une opération I

- Programmation défensive :
  - Nous proposons de vérifier la précondition au début de l'opération et de lever une exception en cas de non-respect de la précondition
- Programmation par assertion :
  - Pour les méthodes qui modifient l'état, nous proposons de vérifier l'invariant par assertion[4] à la fin d'une opération, avant de retourner la valeur de retour
- Tests unitaires :
  - Nous proposons de vérifier dans les tests unitaires
    - la postcondition[5]
    - la levée des exceptions

4. La programmation des assertions et des exceptions sont étudiées lors de la séance 6. En JAVA, cf. l'instruction `assert` : par exemple, « `assert invariant();` »

5. Rappelons qu'une postcondition exprime les conditions à vérifier à l'issue de l'opération, mais uniquement si la précondition est auparavant satisfaite

2.3.7 Précondition, postcondition, et algorithme d'une opération II

P.ex., pré-/post-conditions, et algo. constructeur de Document

- Précondition et invariant

```
constructeur(String c, Localisation l, String t, String au, String an, Genre g)
    si code = null ∨ code = "" alors levée d'une exception
    ... // vérification des autres arguments
    code := c; titre := t; auteur := au; annee := an
    localisation := l; genre := g
    empruntable := false
    emprunté := false
    assert invariant()
```

- Postcondition qui se teste de l'extérieur de la méthode, c'est-à-dire dans les tests unitaires
  - Dans l'exemple ci-avant, « l'objet est effectivement et correctement construit »

2.3.7 Précondition, postcondition, et algorithme d'une opération III

P.ex., pré-/post-conditions, et algo. `emprunter` de la classe `Document`

- Précondition et invariant

```
void emprunter()
    si ¬ (empruntable ∧ ¬emprunté) alors levée d'une exception
    emprunte = true
    genre.emprunter()
    nbEmprunts++
    assert invariant()
```

- Postcondition pour les tests unitaires[6]
  - $\text{empruntable}' \wedge \text{emprunte}' \wedge (\text{nbEmprunts}' = \text{nbEmprunts} + 1)$

6. Le prime exprime l'état à la fin de l'exécution de l'opération : p.ex. $\text{emprunte}'$ est la valeur de l'attribut `emprunte` à la fin de l'opération `emprunter`

3 Préparation des tests unitaires

- Table de décision des opérations
  - Pour les classes avec diagramme de machine à états, on construit les tables de décision : (1) du ou des constructeurs, et (2) des opérations qui modifient l'état de l'objet
  - P.ex, l'opération `emprunter()` de la classe `Document`

	Numéro de test	1	2	3
Précondition	`empruntable`	F	T	T
	`emprunté`		F	T
Postcondition	$\text{nbEmprunts}' = \text{nbEmprunts} + 1$		T	
	$\text{empruntable}' = \textit{true}$		T	
	$\text{emprunte}' = \neg\text{emprunte}$		T	
Exception	Levée d'une exception	OUI	NON	OUI
Effet	Emprunt accepté	F	T	F
	Nombre de jeux de test	1	1	1

  - Rappels :
    - Le non-respect de la précondition est vérifié par la levée d'une exception
    - La postcondition n'est vérifiée que lorsque la précondition est satisfaite

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

- Conception détaillée et préparation des tests unitaires
  - Raffinement du diagramme de classes
  - Fiche de classe
  - Diagramme de machine à états + invariant
  - Tables de décision de tests unitaires
- Rendu de la séance en HP : diagrammes PlantUML + `readme.md`
  - Vérifiez bien que les images `svg` et `png` sont à jour !
- Suivi 2 : principalement conception, et aussi spécification et préparation des tests de validation
- Pour la séance 5 : prérequis « Programmation JAVA » (cours, exercices)

Références I

Cunningham, W. (2009).
TechnicalDebt.
https://www.youtube.com/watch?v=pqeJFYwnkjE.

Fowler, M. (2003).
TechnicalDebt.
http://martinfowler.com/bliki/TechnicalDebt.html.

Giardino, C., Paternoster, N., Unterkalmsteiner, M., Gorschek, T., and Abrahamsson, P. (2016).
Software Development in Startup Companies : The Greenfield Startup Model.
IEEE Transactions on Software Engineering, 42(6) :585–604.

Lamport, L. (2003).
Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers.
Addison Wesley.

Références II

Liskov, B. and Wing, J. (1994).
A Behavioral Notion of Subtyping.
ACM Transactions on Programming Languages and Systems, 16(6) :1811–1841.

OMG (2013).
OMG Unified Modeling Language, Version 2.5.
Specification number formal/2013-09-05.