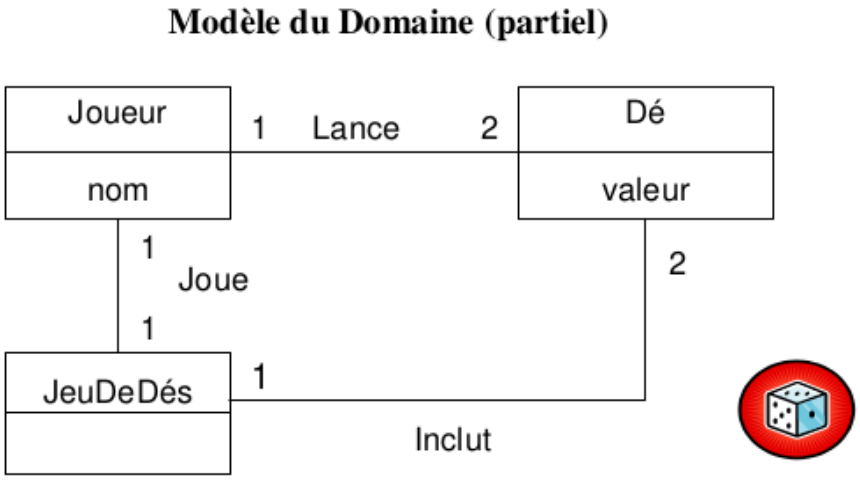
INF1163

Fiche 2

Cours 5 - Modèle du domaine

- Appelé aussi modèle conceptuel
- Artéfact très important de l'analyse OO
- Unified Processus : modélisation métier
- Décrire les concepts du domaine d'application
- Modèle du domaine
  - Concepts
  - Attributs
  - Associations
- Concepts du monde réel
- Pas d'opération
- Vue très statique

Vue statique



Vue dynamique

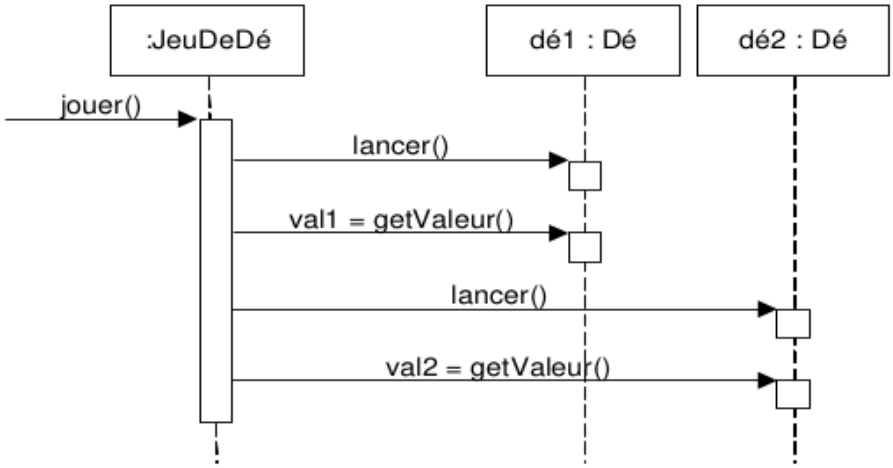
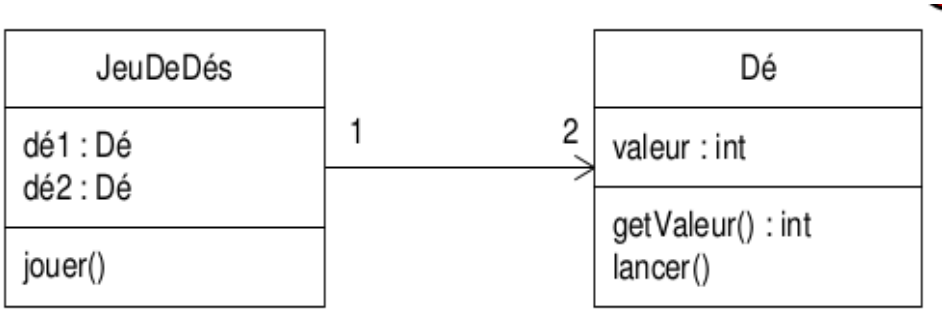


Diagramme de classe



Construction du modèle du domaine

- Identifier les concepts importants et représentatifs du problème
- Dictionnaire visuel du domaine
- On trouve les concepts dans les cas d'utilisation

Identification des concepts

- Utilisation de listes de catégories de classes conceptuelles
    - Liste de référence
    - Considérer la pertinence de chaque catégorie par rapport à notre système
- Identification des groupes nominaux
    - Description textuelle détaillé du domaine du problème
    - Narrations des cas d'utilisation
    - Nom singulier
    - Raisons possibles de l'élimination :
        - Les noms jugés inappropriés par rapport à l'application
        - Redondance : un même concept (synonyme) avec un nom différent (usager, client, …)
        - Flou, vague
        - Evènement ou opération : un nom décrit une action qui n'a pas d'état pertinent
        - Noms trop génériques : ex : système, environnement, …

        - Concepts externes au système : non d'intervenants externes
        - Attribut : nom décrivant une propriété
- Concept : quelques choses de sophistiqués, décrit par plusieurs attributs
- Règle empirique : si un concept X ne nous fait pas penser, de façon impromptue, à un nombre ou à un terme alphanumérique, alors c'est que X est probablement une classe conceptuelle et non un simple attribut
- Écouter des experts du domaine par exemple des logiciels de télécoms

Identification des associations

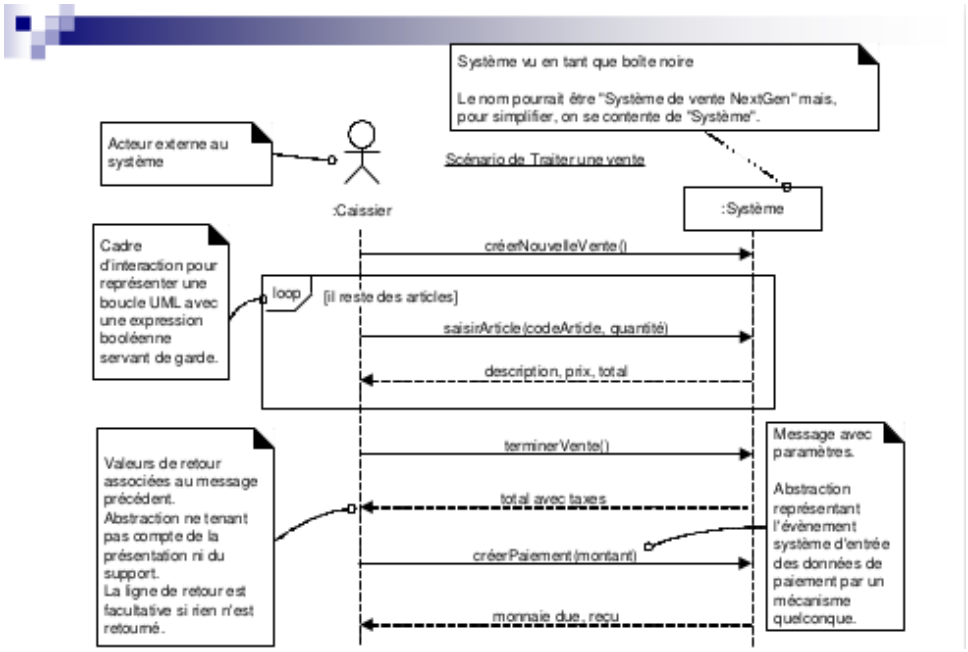
- Utiliser une liste de référence
- Retenir les associations qui améliorent la compréhension du domaine du problème
- Les verbes
- Attentions aux attributs intrus comme les clés étrangères

Définitions

- CRC : Classes-Responsabilités - Collaborations
- Glossaire : liste de termes importants avec définition

Cours 6 – Comportement du système

- Avant de passer à la conception du système, on doit décrire le comportement du système
    - Le quoi
    - Boîte noire
    - Contrats
- DSS – Diagramme de séquence du système
    - Messages échangés entre les acteurs et le système
    - Diagramme du comportement du programme
    - Évènement système = input généré par un acteur et envoyé au système
    - Opération système = opération exécutée par le système en réponse à un évènement système
    - Déroulement d'évènements particuliers tirés d'un cas d'utilisation



- Contrat
    - Document qui décrit les résultats de l'exécution d'une méthode sur l'état des objets du système
    - Préconditions
    - Postconditions
- Responsabilités et méthodes
    - Responsabilité : obligation pour une classe ou un objet d'accomplir quelque chose
        - Faire (doing)
            - Faire quelque chose soi-même
            - Initier une action dans un autre objet
            - Contrôler et coordonner des activités dans d'autres objets
        - Savoir (knowing)
            - Connaître des données privées encapsulées
            - Connaître des objets en relation
            - Connaître des choses qui peuvent être dérivées ou calculées

Conception orienté objet

- Abstraction
- Modularité
- Encapsulation
- Dissimulation des données
- Indépendance fonctionnelle
- Réutilisation

Cours 7 - Patron de conception

- Patron décrit un problème qui se manifeste constamment dans notre environnement
- 4 Eléments
    - Nom : décrit un problème de connexion
    - Problème : description des situations
    - Solution :
    - Conséquences : effets résultant de la mise en œuvre du patron
- Codifier un savoir (ne pas réinventer la roue)

GRASP : General Responsibility Assignment Software Patterns

- Expert
    - Problème : quel est le principe de base le plus important pour affecter une responsabilité à un objet ?
    - Affecter une responsabilité à l'expert en information
    - Bonnes utilisations permettent de plus
        - Comprendre
        - Maintenir
        - Évoluer
- Créateur
    - Qui doit être responsable de nouvelles instances d'une classe donnée ?
    - Affecter à la classe B la responsabilité de créer une instance de la classe A si une ou plusieurs des conditions suivantes sont vraie
        - B contient ou agrège des objets de la classe A

        - B enregistre des objets de la classe A
        - B utilise étroitement des objets de la classe A
        - B contient des données d'initialisation qui seront passées aux objets de type A lors de leur création
- Forte cohésion
    - Garder la complexité à un niveau contrôlable ? Comment s'assurer que les objets restent compréhensibles et faciles à gérer
    - Affecter les responsabilités de façon que la cohésion reste élevée
- Faible couplage
    - Minimiser l'impact d'une modification sur le reste de l'application et assurer une réutilisation élevée des classes
    - Affecter les responsabilités de façon à éviter tout couplage inutile
- Contrôleur
    - Qui est responsable de gérer (recevoir, traiter, …)
    - Affecter la responsabilité de gérer un message de type évènement-système à une classe répondant à l'un des 2 critères
        - Ensemble du système (contrôleur de façade)
        - Gestionnaire artificiel de tous les évènements systèmes (contrôler de cas d'utilisation)

Cours 8 – Conception orienté objet

Visibilité entre objets

- Capacité d'un objet à voir un autre objet, c.a.d avoir une **référence** sur un autre objet.
- Types
    - Attribut
    - Paramètre
    - Locale
    - Globale

Patron de conception : Fabrication Pure

- Problème : à qui affecter une responsabilité lorsque vous voulez préserver une cohésion élevée et un couplage faible et que la solution offerte par le patron de conception expert en informations ne le permet pas
- Solution : affecter un ensemble de responsabilités hautement cohésives à une classe artificielle

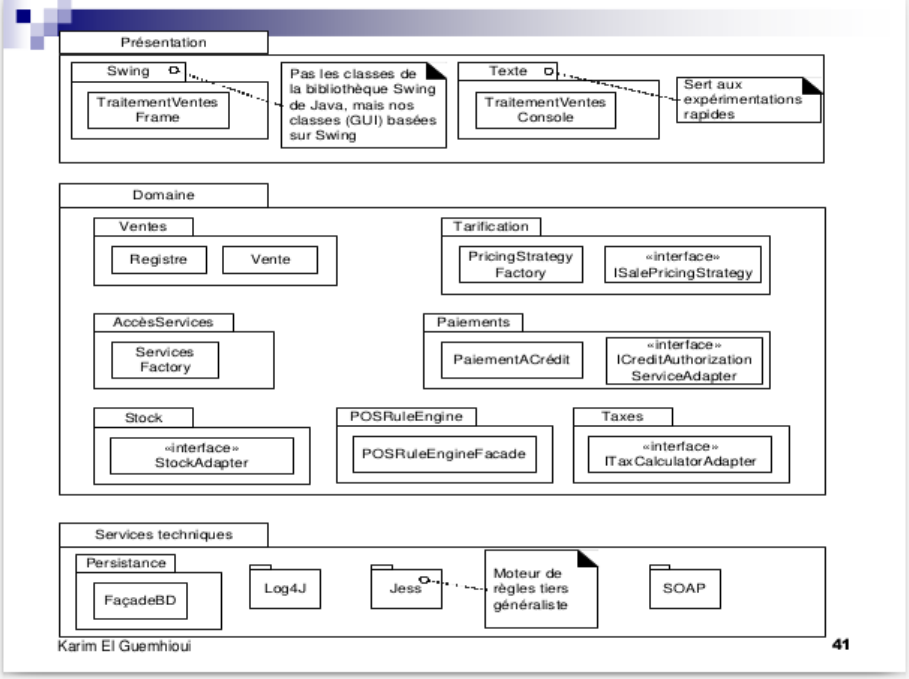
Patron de conception : Singleton

- Problème : seule une instance d'une classe est permise
- Solution : définir une opération de classe qui retourne le singleton

Diagramme de classes

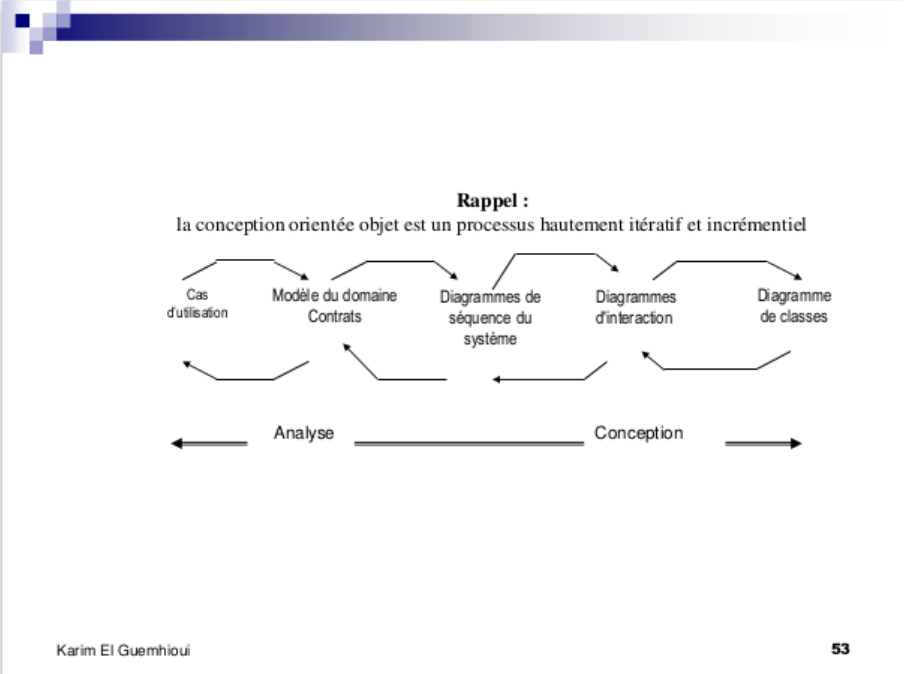
- Quand ?
    - Après avoir créé un modèle du domaine
    - Après diagramme d'interaction

Paquetages et architecture logique



Conception orientée objet en couches revisitée

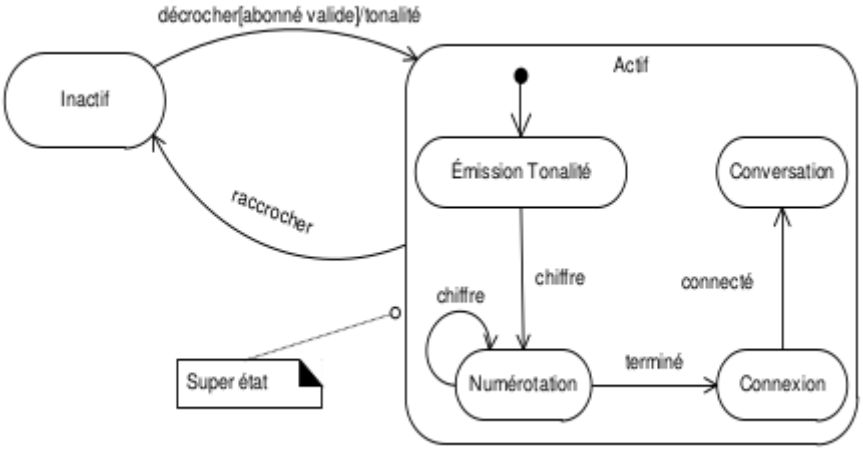
- Séparation des problèmes
- Plusieurs couches
- Réutilisation
- Portabilité
- Maintenance
- Évolutivité



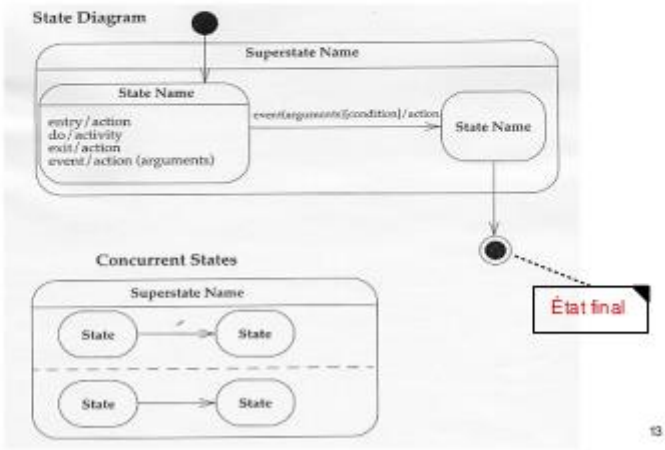
Cours 9 – Modélisation des états et des processus

Modèle d'états

- Décrit les états successifs d'un objet au cours du temps
- Spécifie et implémente les aspects temporels, comportementaux et de contrôle du système
- Modèle d'états est constitué de plusieurs diagrammes d'états.
- Graphe
    - Sommet : états
    - Arêtes : Transition
- Etats : condition d'un objet à un moment donné
- Evènement : objet va changer d'état
- Transition : relation entre 2 états qui indique que l'objet change d'état lorsqu'un évènement se produit



- Des activités à durée finie et interruptibles peuvent être associées à des états (*do-activity*).
- On peut associer à un état une activité à l'entrée ou à la sortie : cette activité sera automatiquement exécutée chaque fois que l'on entrera ou que l'on s'apprêtera à sortir de cet état.



Evénements et changement internet et événements temporels

- Événement de changement interne est causé par la satisfaction d'une expression booléenne et exprimé en utilisant le mot clé when
- Evénement temporel est causé par l'occurrence d'un temps absolu (cas 1) ou par l'écoulement d'une durée (cas 2)

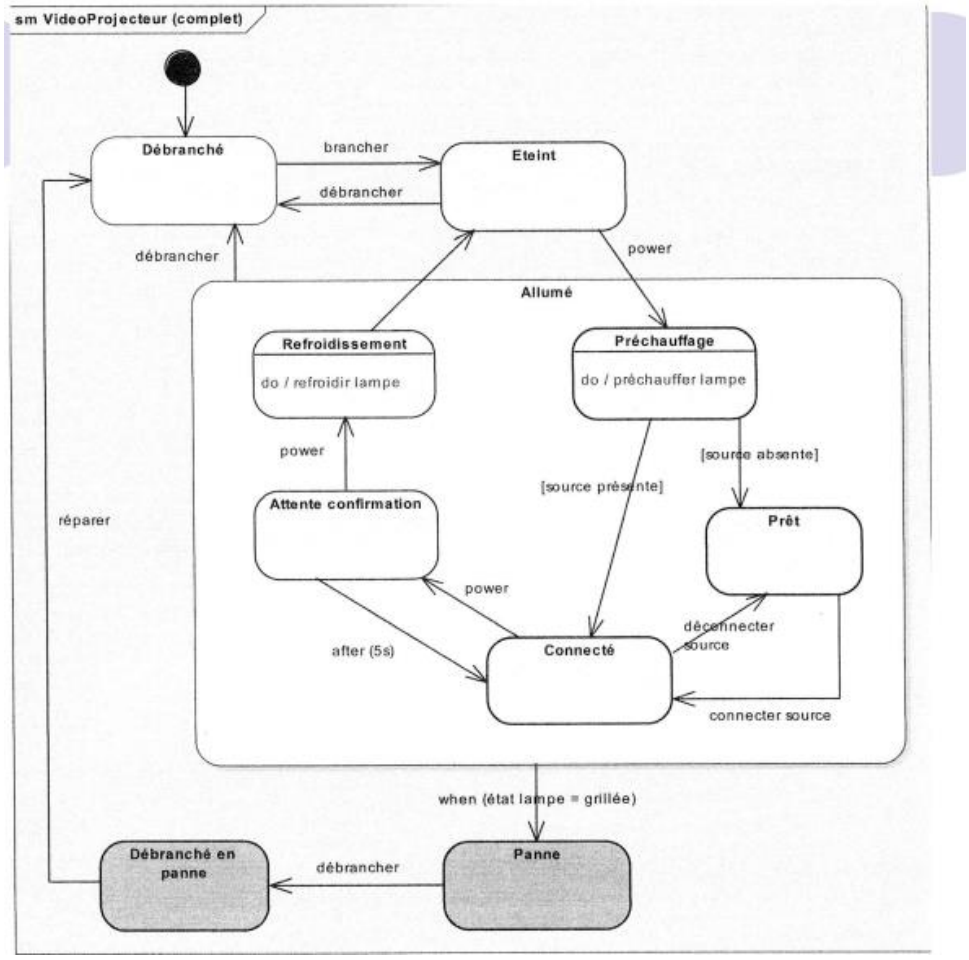
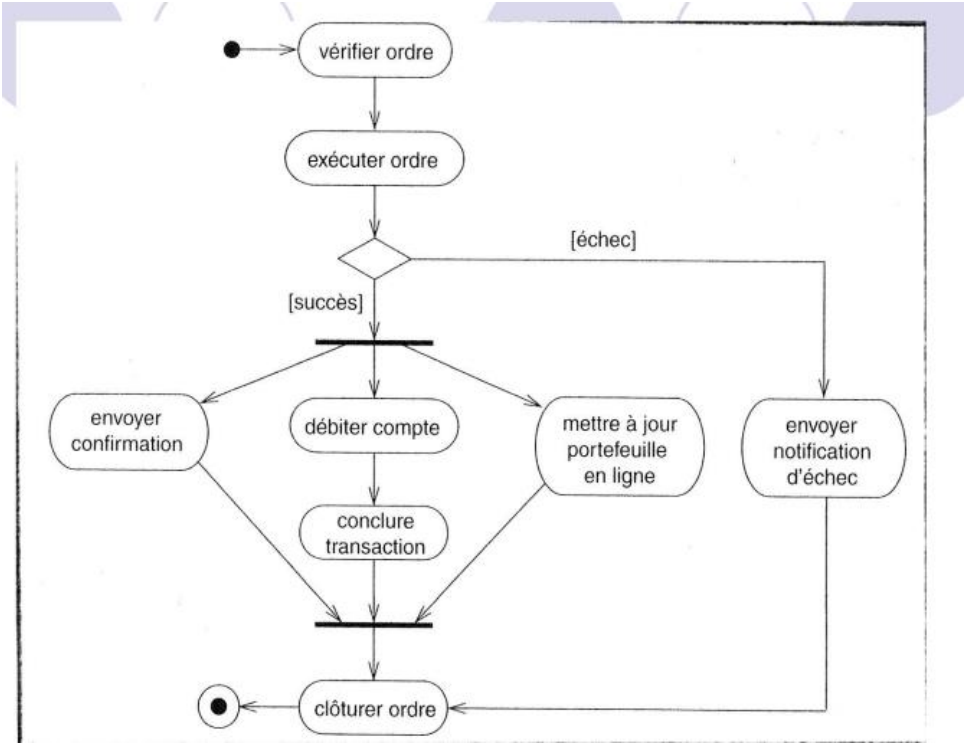
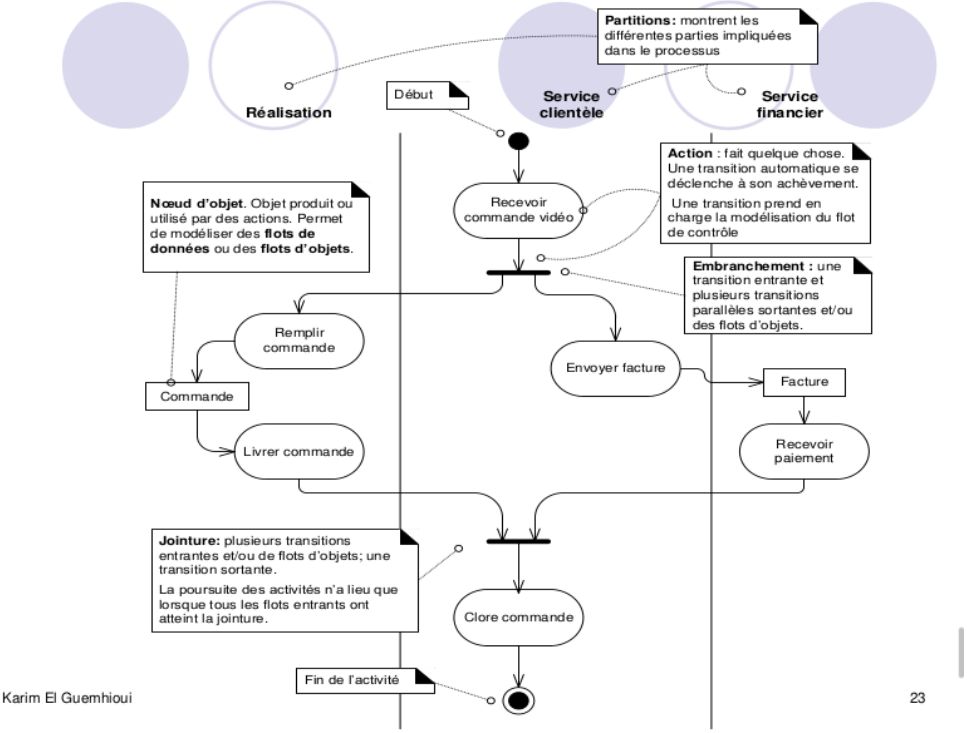


Diagramme d'activités

- Activités séquentielles e parallèles d'un processus
- Modéliser des processus métier, des enchaînements d'activités (workflows)
- Vue dynamique du système
- Flux de contrôle (concentre plus sur les opérations)
- Ordre de transaction





Autres diagrammes statiques de UML

- Diagramme de structure composite (statique)
  - Montre comment une classe complexe est décomposée
- Diagramme de composants (statique)
  - Montre les dépendances entre les différentes parties du code source
- Diagramme de déploiement (statique)
  - Montre les relations physiques entre composants logiciels et composants matériels

- Diagramme global d'interaction (dynamique)
- Diagramme de timing

Cours 10 – Introduction aux tests et à la maintenance

- Tests méthodiques = tests répétables
- Améliorer la qualité du produit
- Faute logicielle est un défaut = erreur du logiciel
    - =! D'une faute hardware

Qualité du logiciel et V & V

- Qualité
    - Degré de satisfaction des besoins du client
    - Fiabilité
    - Performance
    - …
- Vérification
    - Vise à s'assurer que les résultats de chaque phase de développement sont correctement obtenus
- Validation
    - Vise à s'assurer que les résultats de chaque phase de développement traduisent bien les besoins du client

But des tests

- S'assurer que tous les défauts ou bogues sont supprimés
- Trouver le maximum de fautes
- Pas possible de tout tester

Tests unitaires

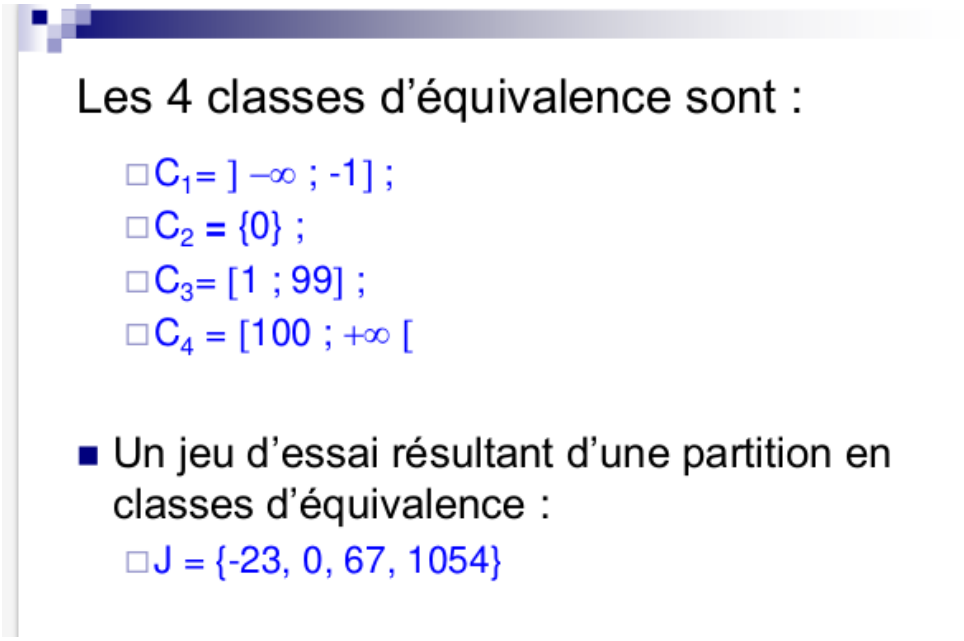
- Tester chaque module individuellement

Boîte noire

- Entrées
- Sorties
- Structure interne du code n'est pas considérés du tout

Techniques fonctionnelles

- Éviter le problème de l'explosion combinatoire du nombre de cas à tester
- Partition en classes d'équivalence
    - Comportement du logiciel devrait être similaire
    - Ensemble des valeurs en entrée génèrent un même comportement de sortie
    - Exemple de la factorielle



a) Le tableau ci-dessous résume les valeurs possibles de l'output en fonction de la valeur de l'input n.

Input n	< -99	-99 à -11	-10	-9 à 9	10	11 à 99	> 99
Output	erreur	positif	impossible	négatif	impossible	positif	erreur

Il en résulte les classes d'équivalence suivantes :

C1 = [-∞ ; -100] ∪ [100 ; +∞ [

C2 = [-99 ; -11] ∪ [11 ; 99]

C3 = [-9 ; 9]

C4 = {-10} ∪ {10}

b) Jeu d'essai minimal : {-2541, 3, 576, 10} (un représentant par classe d'équivalence).

c) {-101, -100, -99, -98, -12, -11, -10, -9, -8, 8, 9, 10, 11, 12, 98, 99, 100, 101}

d) Jeux d'essai supplémentaires en utilisant l'approche heuristique :
- ✓ valeur non-entière (un réel) ;
- ✓ valeur illégale (caractère ou symbole) ;
- ✓ pas de valeur (si c'est possible; cela dépend de comment le programme lit son input).

- Analyse des cas limites
  - Valeurs aux bornes
- Approche heuristique
  - Définit tous les jeux d'essai pour lesquels notre intuition nous dit qu'ils seront efficaces

Boîte transparente

- Connaissance du code source
- Complémentaire par rapport à la précédente
- Différentes formes de tests structuraux
  - Nombre d'instruction couvertes
  - Nombre de branches couvertes
  - Nombre de chemin couverts

Tests de régression

- Jeux d'essai

    - Documenté
    - Sauvegardés de façon électronique
    - Effectuer tous les tests après chaque modification du code

Maintenance

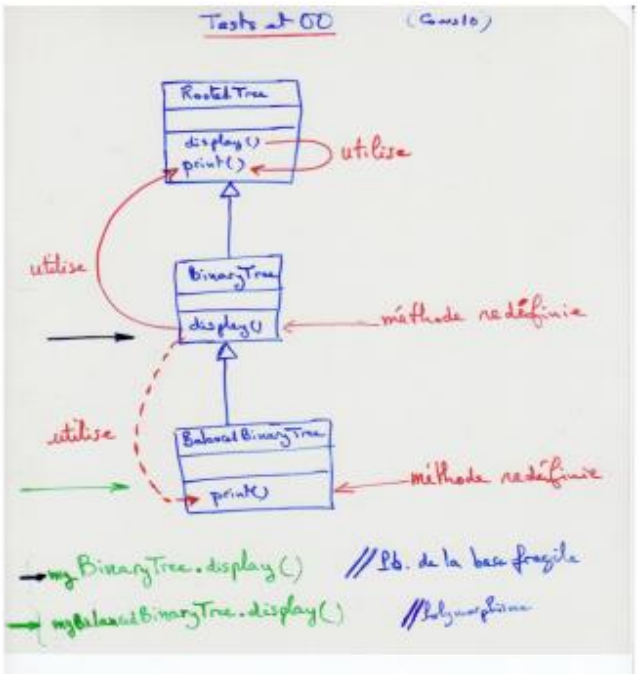
- Toute modification faite à un produit logiciel après sa livraison est considérée comme de la maintenance
- Certains auteurs parlent d'évolution du logiciel
- Logiciel ne s'use pas avec le temps
- Raisons de maintenances
    - Faute s'est manifestée
    - Faute identifiée avant qu'elle ne se manifeste
    - Certains besoins n'ont pas été satisfait
    - Nouveaux besoins
    - Environnement d'exécution a changé
- Catégories
    - Corrective
    - Préventive
    - Perfective
    - Adaptative
- Plusieurs aspects
    - Re-documentation
    - Restructuration
    - Retro-ingénierie
    - Ré-ingénierie

Cours 11 – Techniques de tests suite

Héritage et polymorphisme

- Méthode peut être implémenté différemment
- Surcharge des méthodes

Problème de la classe fragile



d

Développement guidé (ou piloté) par les tests

- Test Driven Design (TDD)
- Tester très tôt dans le processus
- Tests unitaires avant l'écriture du code des modules à tester

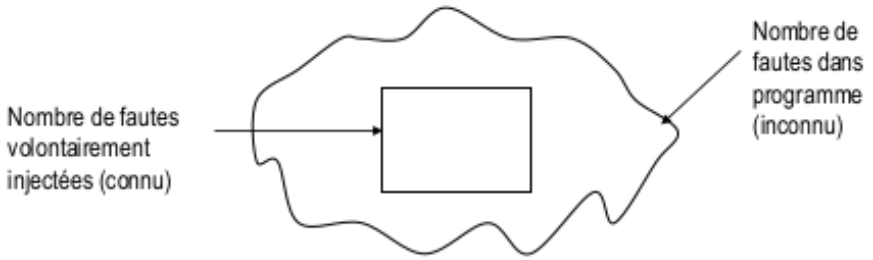
- Elle contraint le code au design:

- Test arrêté à cause de contraintes de temps ou d'argent
- Fixer un objectif à atteindre au niveau de la qualité du produit final
- Plusieurs mesures ont été proposées pour garantir une certaine objectivité

Techniques

- Volume de code couvert
  - 100 % de couverture d'instructions
- Mesures empiriques
  - Critères quantitatifs
  - LoC (nombre de lignes de code source)
  - Complexité cyclomatique de McCabe
- Injection de fautes (fault seeding)
  - Technique Monte Carlo (méthode au hasard)
  - Fautes résiduelles

Technique de Monte Carlo transposée à l'estimation du nombre de fautes résiduelles dans un logiciel



- **Procédure** : une équipe de testeurs, pas au courant des fautes volontairement injectées, teste le programme. Soient :
- n : le nombre de fautes trouvées
- N_i : le nombre de fautes injectées
- n_{it} : le nombre de fautes trouvées parmi les fautes injectées ($n_{it} \leq N_i$)
- N : le nombre de fautes dans le programme (inconnu)
- $$\frac{n}{N} = \frac{n_{it}}{N_i} \Rightarrow N = \frac{n}{n_{it}} \times N_i$$

- Approche intuitive
- Si on a fait 100 fautes faciles et que l'équipe a trouvé d'autres 100 fautes, programme de piètre qualité
- Estime
- Efficacité des tests
- Degré d'achèvement des tests

4) Estimation du nombre de fautes résiduelles

- Cette technique est une alternative à la méthode de Monte-Carlo.
- Elle permet également d'obtenir une estimation du nombre de fautes restant dans un logiciel.
- On recourt à 2 groupes de testeurs

  x : nombre de fautes détectées par le groupe 1

  y : nombre de fautes détectées par le groupe 2

  q : nombre de fautes détectées par les 2 groupes

  (q ≤ x et q ≤ y)

  N : nombre de fautes dans le programme (inconnu)

Solution 4

a) Estimation du nombre de bogues

N_i = 21 fautes injectées
n = 18 fautes trouvées

Selon l'énoncé, sur les 18 fautes trouvées, 2/3 sont des fautes injectées, donc :

n_{it} = 2/3 × 18 = 12 fautes injectées et trouvées

Et donc :
$N = n/n_{it} \times N_i = 18/12 \times 21 = 31,5$ – On arrondit à 32

b) Nombre de bogues résiduels

32 – 18 – (21 – 12) = 5

On peut donc estimer qu'il reste encore 5 bogues à trouver dans ce programme.

a) On a : $E_1 = 90\ \% = 0,9$ et par définition : $E_1 = q/y \Rightarrow y = q/E_1 = 18/0,9 = 20$ fautes.

On a : $y = x-1 \Rightarrow x = y+1$ c.-à-d. x = 21 fautes.

Par définition : $E_2 = q/x = 18/21 = 6/7$

Mais aussi, par définition :

$$E_2 = y/N \Rightarrow N = y/E_2 = 20/(6/7) = (7 \,.\, 20)/6 = 70/3 \simeq 23,33$$

Donc on estimera le nombre de fautes contenues dans le logiciel à 23.

b) Nombre de fautes résiduelles :

$N – y - (x - q) = 23 – 20 – (21 – 18) \simeq 23 – 20 – 3 = 0$

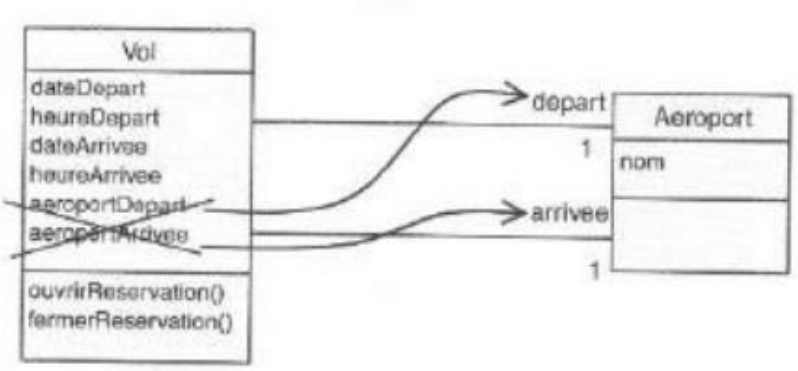
Tests non-exécutables

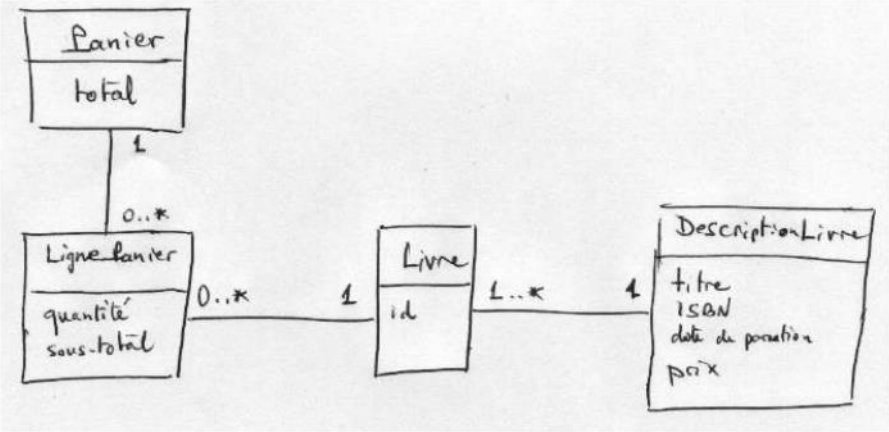
- Tests statiques
- Technique Walkthrough
  - Informelle
  - Petit groupe (4 à 6 personnes)
  - Auteur explique son travail en cheminant pas à pas à travers ses livrables
- Technique de l'inspection
  - Plus formelle
  - Ordre du jour et documents distribués à l'avance
  - 4 à 6 participants
  - Inspection du travail sur la base
  - La liste de contrôle (inspection list) préparée à l'avance, contient les types de fautes caractéristiques pour le genre de travail à inspecter
- Applique

Preuves formelles (Correctness Proof)

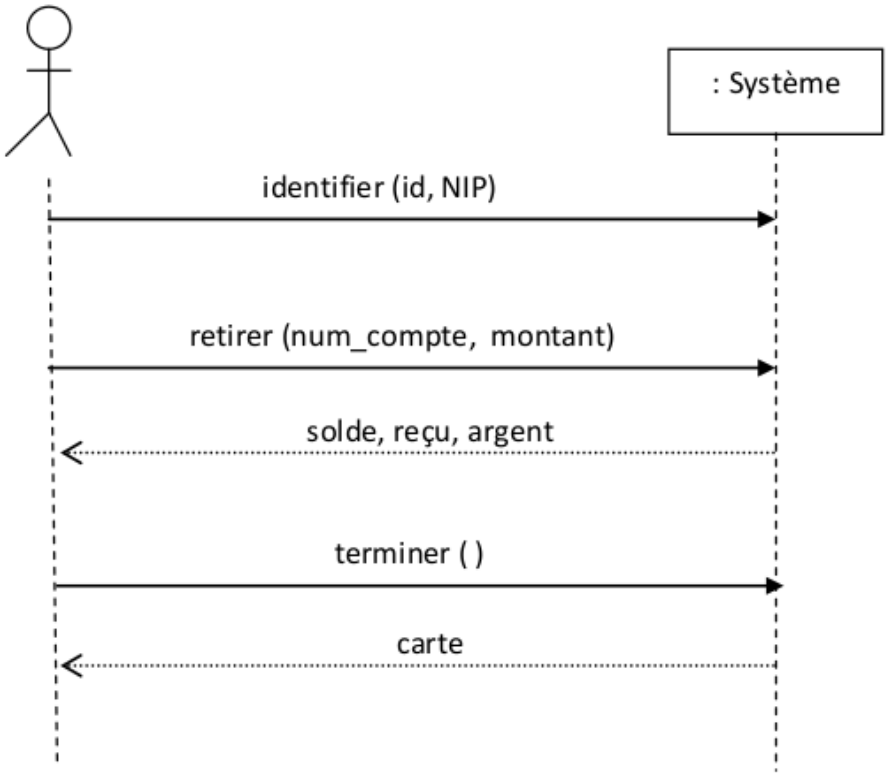
- Technique mathématique
- Prouver un logiciel est conforme à sa spécification
- Approche axiomatique
- Plusieurs conditions nécessaires
    - Sémantique du langage de programmation doit être définie de façon formelle
    - Programmation doit être spécifié de façon formelle dans une notation compatible avec la technique de vérification mathématique utilisée

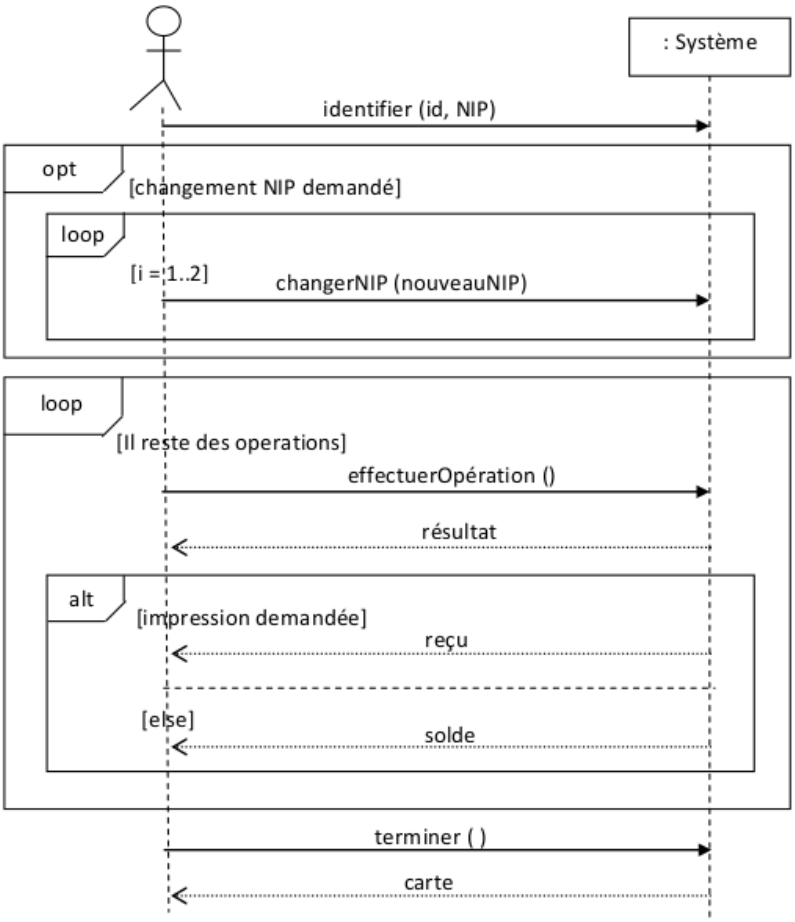
TD2 – Modèle du domaine





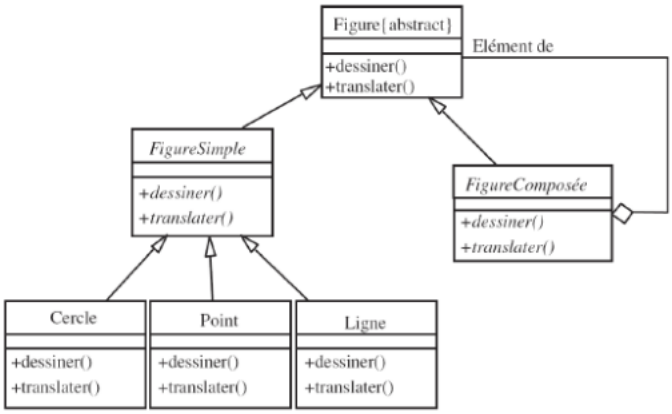
TD3 - Modélisation et conception orientée objet



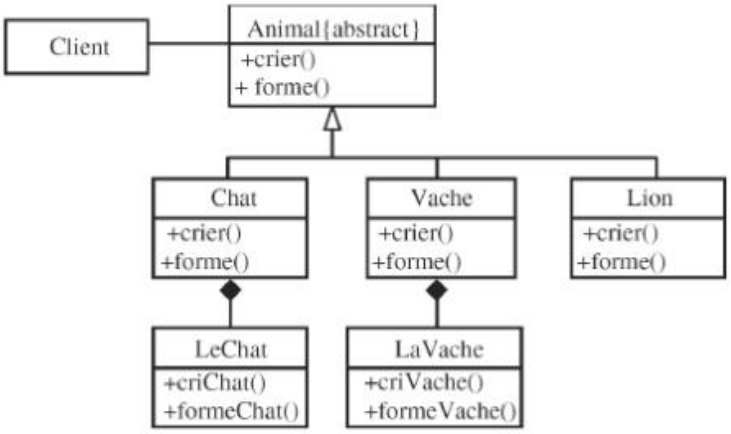


TD4 – Patron de conception

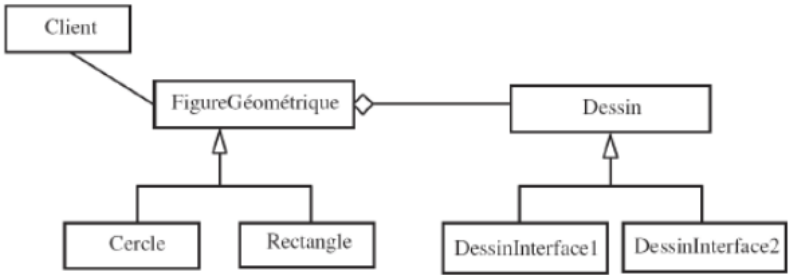
Patron composite



Patron adapter



Patron bridge



Patron Observer