Génie Logiciel Avancé
Cours 1 — Mise à niveau GL6

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2014–2015

URL http://upsilon.cc/zack/teaching/1415/gla/
Copyright © 2011–2015 Stefano Zacchiroli
© 2014 Mihaela Sighireanu
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

Génie Logiciel
- une introduction

Mihaela Sighireanu

sighirea@liafa.univ-paris-diderot.fr

Qu'est-ce que le génie logiciel ?

- Le **génie logiciel** est un domaine des sciences de l'ingénieur dont l'objet d'étude est
  - la conception,
  - la fabrication et
  - la maintenance

des systèmes logiciels.

Qu'est-ce qu'un système ?

- Un **système** est un ensemble d'*éléments interagissant entre eux* suivant un certains nombres de principes et de *règles dans le* but de réaliser un *objectif*.

- Caractéristiques des systèmes :
  - Possède une frontière (= critère d'appartenance au système)
  - Evolue dans un environnement (= la partie du monde extérieure au système)
  - Hiérarchisé et plusieurs sous-systèmes
  - Complexité = dimension + évolutivité

Qu'est-ce qu'un logiciel ?

- Un **logiciel** est un ensemble d'entités nécessaires au fonctionnement d'un processus de traitement automatique de l'information.
- Exemples d'entités :
  - des programmes (en format *code source ou exécutables) ;*
  - des documentations d'utilisation ;
  - des informations de configuration.
- Un logiciel est un (sous-)système :
  - Il interagit avec des clients (humain, logiciel, matériel)
  - Il réalise une **spécification** : son comportement vérifie un ensemble de critères qui régissent ses interactions avec son environnement.

Objectifs du génie logiciel

- Le génie logiciel vise à garantir que :
  - la spécification du système logiciel **répond** aux **besoins** réels de ses clients :
    - identifier les besoins
    - concevoir des solutions adaptées
    - démontrer que les besoins sont satisfaits
  - le logiciel respecte sa spécification :
    - valider le logiciel par rapport à sa spécification
  - les coûts et les délais de réalisation sont respectés :
    - « time is money »

Comment spécifier un logiciel ?

- Répondre à la question

  Que doit faire le logiciel ?

- La spécification d'un logiciel peut prendre de nombreuses formes :
  - formelles : logique, automate, théorie des ensembles, …
  - semi-formelles : diagramme, fiches structurées, …
  - informelles : texte en langage naturel, prototype, …
- La *complexité et la taille* de la spécification peuvent varier énormément en fonction de l'environnement d'utilisation du logiciel et des objectifs auxquels il répond.

Exemple de spécifications

- Module logiciel d'aide à la rédaction de SMS
  - Informelle : l'utilisateur qui saisit un mot reçoit une liste de mots qui peuvent compléter le début de mot saisi.
    - La taille de cette liste ?
    - Dans quel dictionnaire ?
    - Comment apparaissent les choix ?
  - Semi-formelle :
    - Pre-condition : l'utilisateur introduit une séquence s de lettres de longueur minimum 2; le système dispose d'un dictionnaire.
    - Post-condition : une liste de 5 mots qui ont comme commencent avec s et qui se trouvent dans le dictionnaire est affichée dans l'ordre alphabétique.

Exemple de spécifications

- Module logiciel d'aide à la rédaction de SMS
  - Spécification semi-formelle : avec diagrammes UML
    - Diagrammes de cas d'utilisation
    - Diagrammes de classes
    - Diagrammes d'intéraction = diagrammes séquences
    - Diagrammes d'état-transition = automates
    - Diagrammes d'activités = flot de contrôle

Exemple de spécifications

- Module logiciel d'aide à la rédaction de SMS
  - Formelle : en utilisant des contrats dans la logique de Hoare
    - Pre-condition : s is a parameter of type string

      □ (s ≠ null & len(s)≥2)

      Où

      type string = char*

      len : string ➔ nat

      null : string where len(null) = 0
    - Post-condition : result is of type string*

      □ (len(result) ≤ 5) & (forall i. 0 <= i < len(result) =>prefix(s,get(result,i))) & (forall i,j 0<= i < j < len(result) => get(result,i) < get(result,j))

      Où

      get : (s: ty*, i: nat) -> ty provided that s != null & i < len(s))

      prefix : (p:ty*, s:ty*) -> bool

      < : (l: string*, g:string*) -> bool

Comment fabriquer un logiciel de qualité ?

- En plus du respect de sa spécification, la **qualité** d'un logiciel dépend des 4 critères suivants :
  - **Maintenabilité** : Peut-on faire évoluer le logiciel ?
  - **Robustesse** : Le logiciel est-il sujet à des dysfonctionnements ?
  - **Efficacité** : Le logiciel fait-il bon usage de ses ressources ?
  - **Utilisabilité** : Est-il facile à utiliser ?

Comment fabriquer un logiciel de qualité ?

- Pour répondre à cette exigence de qualité, on a essayé d'appliquer les **méthodes** connues de l'ingénieur au domaine du logiciel, pour établir des **méthodes** fiables pour le logiciel.

- Méthode = un cadre rigoureux pour :
  - Guider le développement du logiciel, de sa conception à sa livraison.
  - Contrôler les coûts, évaluer les risques et respecter les délais.
  - Établir des critères d'évaluation de la qualité d'un logiciel.

Comment fabriquer un logiciel de qualité ?

- Cependant, la construction d'un logiciel diffère de celle d'un pont …
  - les progrès technologiques très rapides peuvent rendre un logiciel caduque
  - on ne sait pas très bien réutiliser les programmes existants
- Le génie logiciel est un domaine en pleine évolution qui offre une grande palette d'outils et de méthodes pour parvenir à construire du logiciel de qualité.
- Aucune de ses méthodes ne s'est imposée à ce jour, donc du recul est nécessaire !
- Ces méthodes se distinguent principalement par :
  - *leur degré de formalisme ;*
  - *leur champ d'application ;*
  - *les contraintes de qualité qu'elles ambitionnent.*

Méthodes de conception du logiciel

- Méthodes formelles : B, Model-checking, SPARK, …
  - utilisent des outils mathématiques et des méthodes de preuve pour construire un logiciel correct par construction dont la vérification est automatisée ou assistée.
- Méthodes sémi-formelles : Rational Unified P., Merise, …
  - visent à introduire un langage normalisé pour décrire le logiciel et sa spécification, mais ces langages contiennent certaines ambiguïtés et n'offrent aucune garantie sur la qualité des résultats.
- Méthodes empiriques : test unitaire, peer reviewing, …
  - un ensemble de « bonnes pratiques » qui ont fait leur preuve par l'expérience

Les grands principes du génie logiciel

- Un certain nombre de grands principes (de bon sens) se retrouvent dans toutes ces méthodes. En voici une liste proposée par Ghezzi :
  1. La rigueur
  2. La décomposition des problèmes en sous-problèmes indépendants
  3. La modularité
  4. L'abstraction
  5. L'anticipation des évolutions
  6. La généricité
  7. La construction incrémentale

Les grands principes du génie logiciel

1. La rigueur :
   - Les principales sources de défaillances d'un logiciel sont d'origine humaine.
   - Des outils de vérification accompagnant le développement peuvent aider à réduire les erreurs. Cette famille d'outils s'appelle CASE (*Computer Aided Software Engineering*).
     - *Typeurs, générateurs de code, générateurs de tests, assistants de preuve, …*
2. La décomposition en sous-problèmes :
   - Simplifier les problèmes (temporairement) pour aborder leur complexité progressivement.
   - Décorréler les problèmes pour n'en traiter qu'un seul à la fois.

Les grands principes du génie logiciel

3. La modularité :

- Partitionner le logiciel en modules qui :
  - *ont une cohérence interne (des invariants)*
  - *possèdent une interface ne divulguant sur le contenu du module que ce qui est nécessaires aux modules clients*
  - *réalisent l'indépendence de l'évolution*

4. L'abstraction :

- Exhiber les concepts généraux qui englobent un certains nombre de cas particuliers.
  - Classes abstraites, polymorphisme, templates, généricité

5. L'anticipation des évolutions :

- Concevoir un système suffisamment riche pour que l'on puisse le modifier incrémentalement.

Les grands principes du génie logiciel

6. La généricité :

- Un logiciel réutilisable a beaucoup plus de valeur qu'un composant dédié.
- Un composant est générique lorsqu'il est adaptable.

7. La construction incrémentale :

- Un développement logiciel a plus de chances d'aboutir si il suit une cheminement incrémental (*baby-steps).*

☑ Ecrire le code source d'une fonction ou module, le compiler, et passer à la suivante

☹ Écrire l'ensemble du code source d'un programme et compiler.

Le processus de développement logiciel

- Un **processus de développement logiciel** est un ensemble *structuré* d'activités que conduisent à la production d'un logiciel.

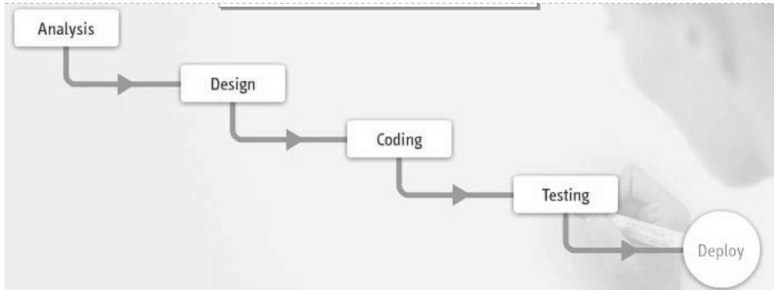
- Il n'existe pas de processus idéal, mais les **activités** des processus existants se regroupent en 6 grandes catégories :
  1. La spécification *(analysis)* du logiciel définit ses fonctionnalités et leurs contraintes.
  2. La conception *(design)* . . .
  3. . . . et l'implémentation *(coding)* sont chargées de réaliser le logiciel, en conformité avec sa spécification.
  4. La validation s'assure effectivement du respect de la spécification par le logiciel produit.
  5. La livraison *(deployement)* installe le logiciel chez les clients.
  6. L'évolution adapte le logiciel aux besoins futurs de ses clients.

Le processus de développement

- Il est très rare d'appliquer un processus comme une unique séquence des 6 activités précédentes.
- En général, un logiciel complet est le fruit de plusieurs itérations, chaque itération contient les 6 activités.

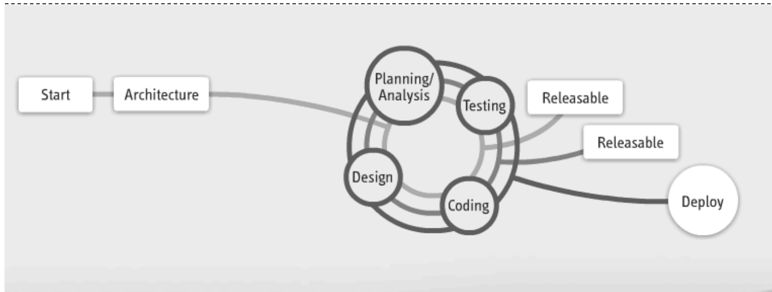
- Il existe différents modèles de processus qui organisent de façon différente ces activités, entre eux :
  - *le modèle en cascade (warterfall) ,*
  - *le modèle de développement évolutif (iterative) et*
  - *le modèle de développement par composants.*

Modèle en cascade



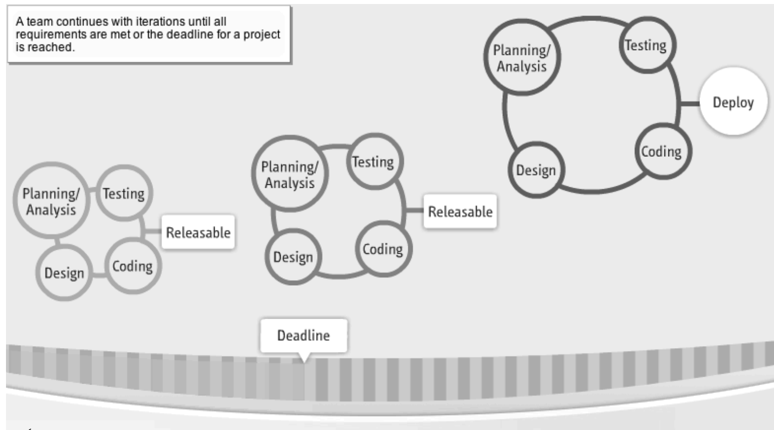
- Chaque phase doit se terminer pour commencer la suivante.
- Des documents sont produits pour concrétiser la réalisation de chaque phase.
- Il hérité des méthodes classiques d'ingénierie, il s'adapte donc bien dans un contexte où le logiciel fait partie d'un système complexe englobant, mais **inadapté** au développement de systèmes dont la spécification est difficile à formuler *a priori*..

Modèle évolutif



✓Un prototype est écrit rapidement et est confronté à l'utilisateur.
✓En fonction du résultat, on raffine la spécification.
✓On reprend le prototype ou on le réécrit jusqu'à l'obtention du système final.
✓Il augmente les chances de répondre aux besoins de l'utilisateur car il permet de les comprendre plus rapidement : voir le « manifeste » de la méthode agile
✓Il ne dispense d'écrire la spécification du système car il faut s'assurer que l'implémentation est correcte.

Modèle évolutif



- ✓Il est adapté aux projets de taille moyenne.
- ✓Critique : il est difficile de structurer correctement le logiciel (définir de bonnes abstractions, modulariser efficacement) et ne pas faire des produits « bricolés ».

Génie Logiciel
- une introduction

Cours 4 : UML, diagrammes de cas d'utilisation, de séquences de messages

Qu'est-ce que UML ?

- **UML** (*Unified Modeling Language*) est une notation graphique standardisée par l'OMG *(Object Management Group)* pour
  - Représenter
  - Spécifier
  - Construire
  - Documenter

les systèmes logiciels dont les objectifs sont :

  - La promotion des techniques « orientées objet »
  - Être compréhensible par les humains (informaticiens ou non) et interprétable par la machine

Quand utiliser UML ?

- Dans la phase d’analyse et conception préliminaire
  - Pour spécifier les fonctionnalités attendues
    - Diagrammes de cas d’utilisation et de séquences
  - Pour décrire l’architecture du système
    - Diagrammes de classe et de paquetage
  - Pour spécifier/coder le comportement des systèmes
    - Diagrammes d’états (automates) et d’activités (réseaux de Petri)

UML n’est pas …

- Une méthode de développement, c’est à dire une démarche à suivre lors de la conception ou la réalisation du système.
  - Des processus de développement basés sur UML existent :
    - RUP (*Rational Unified Process*)
      - Approche itérative où le développement est divisé en étape qui doivent être maitrisées
    - MDA (*Model Driven Architecture*)
      - Sépare la logique du métier de toute plate-forme technique en utilisant un modèle métier indépendant de la plate-forme technique à partir duquel on génère automatiquement du code vers la plate-forme choisie. L’élément central n’est plus l’objet mais le modèle.
- Un langage de programmation
    - Des outils existent qui génèrent du code à partir de certaines diagrammes UML (classe, paquetage, activités)
- Un langage formel
    - La sémantique de la notation n’est pas toujours claire

Vue globale des diagrammes UML

Diagramme

Diagramme Structurel

Diagramme Comportemental

Diagramme de Classes

Diagramme de Composants

Diagramme d'Objets

Diagramme de Structure Composite

Diagramme de Déploiement

Diagramme de Packages

Diagramme d'Activités

Diagramme de Cas d'Utilisation

Diagramme d'Etats

Diagramme d'Interaction

Diagramme de Séquence

Diagramme de Vue Globale d'Interaction

Diagramme de Communication

Diagramme de Timing

Vue des diagrammes UML par domaine

- Domaine structurel
  - Vue fonctionnelle : diagrammes de cas d'utilisation
    - conceptualise et structure les besoins de l'utilisateur, sert d'interface entre les différents intervenants du projet
    - définit le contour du système et ses principales fonctionnalités
  - Vue statique : diagrammes de classe et d'objets
    - définit les principaux classeurs de l'application et leur relation (composition, généralisation, etc.)
  - Vue conceptuelle : dia. de collaboration et de composants
    - définit les collaborations dynamiques des classeurs et l'architecture physique du système
- Domaine dynamique
  - Regroupe l'ensemble de vue de comportement du système à l'aide de
  - diagrammes d'activités, d'états, de séquences, de collaboration
- Domaine physique
  - Vue de déploiement : diagrammes de déploiement
    - pour les applications distribuées, définit l'emplacement physique des composants
- Domaine gestion du modèle
  - Montre l'organisation en paquetage du modèle lui même via diag. de paquetage.

Utilisation des diagrammes UML

- Par rapport aux 6 **activités** du développement :
  1. La spécification *(analysis)* définit les fonctionnalités du logiciel
     - Diagrammes de cas d'utilisation, de séquences
  2. La conception *(design)* . . .
     - Diagrammes de classe, de paquetage, d'objet, de composants
  3. L'implémentation *(coding)*
     - Diagrammes d'activités, d'état
  4. La validation
     - Diagrammes de séquence (interprétées comme des cas de test)
  5. La livraison *(deployement)*
     - Diagrammes de déploiement
  6. L'évolution

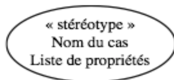
Contraintes en UML : langage OCL

- Le langage OCL (*Object Constraint Language*) est un permettant de **spécifier** des contraintes sur les diagrammes UML
  - En utilisant un langage logique de type logique de Hoare
  - Les spécifications OCL apportent des précisions sur les diagrammes UML (de tout type) sous la forme de pré-conditions, des post-conditions ou des invariants pour une opération
  - Peuvent être exécutés par des simulateurs de modèles UML

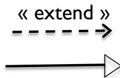
Diagrammes de cas d’utilisation

- Utilisées pour exprimer le besoin des utilisateurs du système, donnent une vision utilisateur de l’application.
- Elements :
  - Acteur = une chose (humain, périphériques, systèmes logiciel) qui interagit avec le système

  - Cas d’utilisation = une fonctionnalité visible de l’extérieur, il réalise un service de bout en bout



  - Relation d’association
    - entre l’acteur et le cas d’utilisation — action
    - entre cas d’utilisation : inclusion, extension — « inclut »
    - acteur-acteur ou cas-cas : généralisation/spécialisation



Diagrammes de cas d'utilisation

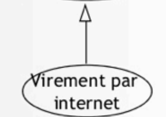
- Eléments
  - Classeur de cas = définit la frontière du systèm

  - Classeur de diagramme = nome le diagramme, mot clé **uc**
  - Commentaires

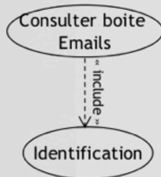
- Sémantique des associations
  - A « include »➔ B : A contient le comportement défini par B
  - A « extend »➔ B : A augmente le comportement de B dans le point d'extension prévu pour B
  - A —▷ B : A est un cas spécifique de B (B est une généralisation de A)

Exemples d’associations

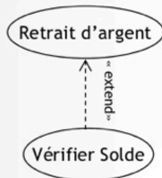
Exemple Généralisation/Spécialisation



Exemple Inclusion



Exemple Extension



Exemples : restaurant

Classeur des cas d'utilisation

uc Use Cases

System Boundary

Waiter

receive order

Order Food

<<extend>>

Order Wine

confirm order

place order

Serve Food

Cook Food

Chef

<<extend>>

{if wine was ordered}

Serve Wine

Client

Eat Food

<<extend>>

{if wine was served}

Drink Wine

facilitate payment

pay

Cashier

accept payment

Pay for Food

<<extend>>

{if wine was consumed}

Pay for Wine

Cas d'utilisation interne

Condition d'extension

Exemple :

Borne interactive d'une banque

Retirer argent

Effectuer un virement

Point d'extension : vérificationSolde {après avoir demandé le montant}

« inclut »

« inclut »

« étend »

S'authentifier

Client

Vérifier le solde

Condition : {si montant > 20 euros}
Point d'extension : vérificationSolde

« inclut »

Consulter comptes

Consulter sur Internet

Cas d'utilis. avec un point d'extension

Cas d'utilisation interne

Commentaire condition

Comment définit les cas d'utilisation

- Quels sont les acteurs ?
  - Humais, périphériques, systèmes logiciel externes …
- Comment ces acteurs se servent du système ?
  - Dans quels cas ils l'utilisent, comment ils s'en servent …
  - Eviter les redondances, choisir un haut niveau d'abstraction, factoriser les cas en utilisant les relations
- Quel est l'enchainement temporel du cas ?
  - Pas dans les diagrammes de cas
  - Utiliser les diagrammes de séquence ou les description textuelles structurés

Description d'un cas d'utilisation

- L'appel par l'acteur d'un cas d'utilisation obéit à des scénarios bien précis
    - Scénario primaire = déroulement basique du cas d'utilisation
    - Scénario secondaire = les cas alternatifs au scenario primaire
        - Actions possible mais pas prises en compte,
        - Traitement des erreurs arrivées pendant le déroulement basique
- Les scénarios d'utilisations peuvent être décrits par
    - Une description textuelle structurée
    - Diagrammes de séquence UML

Description textuelle structurée

Description d'un retrait d'argent

Identification

Nom du cas : retrait d'espèces en euros.
But : détaille les étapes permettant à un guichetier d'effectuer l'opération de retrait d'euros demandé par un client.
Acteur principal : Guichetier.
Acteur secondaire : Système central.
Date : le 18/02/2005.
Responsable : M. Dupont.
Version : 1.0.

Séquencement

Le cas d'utilisation commence lorsqu'un client demande le retrait d'espèces en euros.
Pré-conditions
Le client possède un compte (donne son numéro de compte).
Enchaînement nominal

1. Le guichetier saisit le numéro de compte client.
2. L'application valide le compte auprès du système central.
3. L'application demande le type d'opération au guichetier.
4. Le guichetier sélectionne un retrait d'espèces de 200 euros.
5. L'application demande au système central de débiter le compte.
6. Le système notifie au guichetier qu'il peut délivrer le montant demandé.

Post-conditions
Le guichetier ferme le compte.
Le client récupère l'argent.

Rubriques optionnelles

Contraintes non fonctionnelles
Fiabilité : les accès doivent être extrêmement sûrs et sécurisés.
Confidentialité : les informations concernant le client ne doivent pas être divulguées.
Contraintes liées à l'interface homme-machine
Donner la possibilité d'accéder aux autres comptes du client.
Toujours demander la validation des opérations de retrait.

Description des scénarios secondaires

Description du cas « Rechercher une vidéo »

Identification
Nom du cas : « Rechercher une vidéo ».
But : décrire les étapes permettant au client de rechercher une vidéo *via* le distributeur automatique.
Acteur principal : néant (cas interne inclus dans le cas « Emprunter une vidéo »).
Acteur secondaire : néant.
Date de création : le 31/12/2004.
Responsable : M. Dupont.
Version : 1.0.

Séquencement
Le cas démarre au point 3 de la description du cas « Emprunter une vidéo ».

Enchaînement nominal (le choix du film se fait par genres)

1. Le système demande au client quels sont ses critères de recherche pour un film (les choix possibles sont : par titres ou par genres de film).
2. Le client choisit une recherche par genres.
3. Le système recherche les différents genres de film présents dans le distributeur.
4. Le système affiche une liste des genres (les choix possibles sont action, aventure, comédie et drame).
5. Le client choisit un genre de film.
6. Le système affiche la liste de tous les films du genre choisi présents dans le distributeur.
7. Le client sélectionne un film.

Enchaînements alternatifs

A1 : Le client choisit une recherche par titres.
L'enchaînement démarre après le point 1 de la séquence nominale :

2. Le client choisit une recherche par titres.
3. Le système affiche la liste de tous les films classés par ordre alphabétique des titres.

La séquence nominale reprend au point 7.

Enchaînements d'exception

E1 : Le client annule la recherche.
L'enchaînement peut démarrer aux points 2, 5 et 7 de la séquence nominale :
Appel de l'exception E4 du cas « Emprunter une vidéo ».

Post-conditions
Le système a mémorisé le film choisi par le client.

Rubriques optionnelles

Contraintes non fonctionnelles

Contraintes liées à l'interface homme-machine
Quand une liste de films s'affiche, le client peut trier la liste par titres ou par dates de sortie en salles.
Le client peut se déplacer dans la liste et la parcourir de haut en bas et de bas en haut.
Ne pas afficher plus de 10 films à la fois dans la liste.

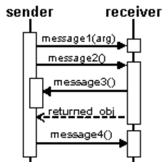
Diagrammes de séquence

- Décrivent comment les acteurs interagissent avec le système
  - Inspirés MSC = « Message Sequence Charts » norme IUT
  - En fonction du détail du système
    - En « boîte noire » : les composants du système de sont pas détaillés
      - Pour l'analyse de besoins (cahier de charges ou spécification fonctionnelle)
    - En « boîte blanche » : les messages internes du système sont détaillés
      - Pour la conception
- Eléments :
  - Classeur : regroupe les éléments du diagramme, mot clé **sd**
  - Ligne de vie : représente un participant à l'interaction
    - Avec/sans activation
  - Messages : représente un échange effectué entre participants
    - Synchrones, asynchrones, perdu, généré
  - Etiquettes de message : nomment et identifient les messages
    - Avec/sans condition
  - Composition de diagrammes : facilitent la réutilisation

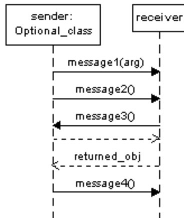
Eléments des diagrammes de séquence

Lignes de vies :

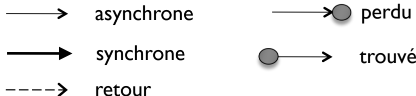
- avec activation :



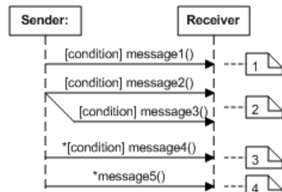
- sans activation :



Messages :



Etiquettes de messages :



1: avec condition
2: avec alternatives (xor)
3: tant que condition
4: avec répétition

f(x)	msg avec argume
f(-)	msg avec arg. non-défini
y = f(x)	retour de f(x), résultat en y
f(x) : v	retour de f(x), valeur en v

Exemple : pompe à essence

client

: Système Pompe

Prendre pistolet

Demander montant

Saisir le montant

Vérification du montant

Montant valide

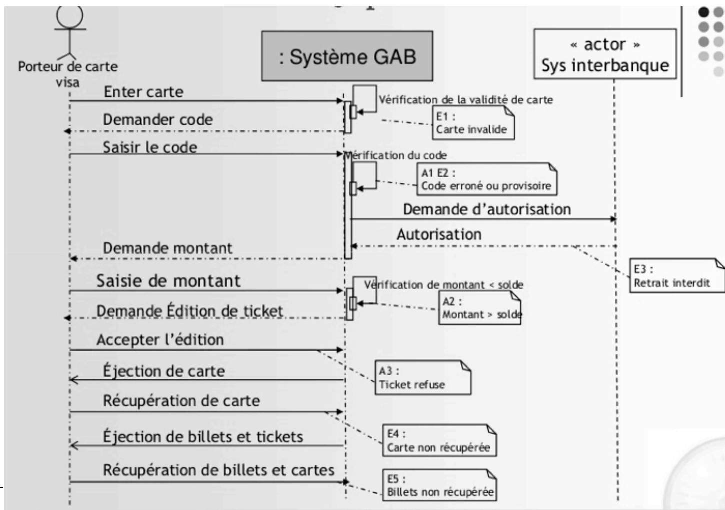
A1 :
Montant invalide

Appuyer sur gâchette

Pomper Essence

Rendre pistolet

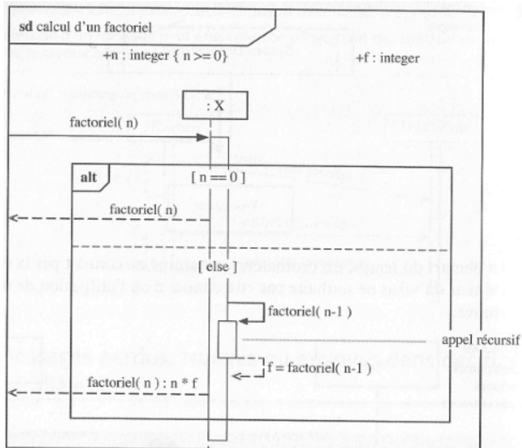
Exemple : terminal bancaire



Composition de diagrammes

- Pour éviter la surcharge en message et pour faciliter la réutilisation, les diagrammes peuvent être composées en utilisant les opérateurs suivants :
  - **ref** = inclusion (référence à) un diagramme
  - **alt** = alternative (if) chaque partie préfixée d'une condition
  - **opt** = alt avec choix unique
  - **assert** = contrainte sur une ligne de vie
  - **loop[(min,Max)]** = itération bornée ou non bornée (loop)
  - **neg** = séquence interdite/invalide
  - **ignore/consider** = lignes de vie/messages à ignorer/prendre en compte
  - **par** = exécution parallèle
  - **critical** = exécution atomique
  - **strict** = exécution en séquence

Exemple : calcul factoriel



Exemple : décollage avion

sd Décollage d'un avion

: TourDeContrôle

: Pilote

: Avion

strict

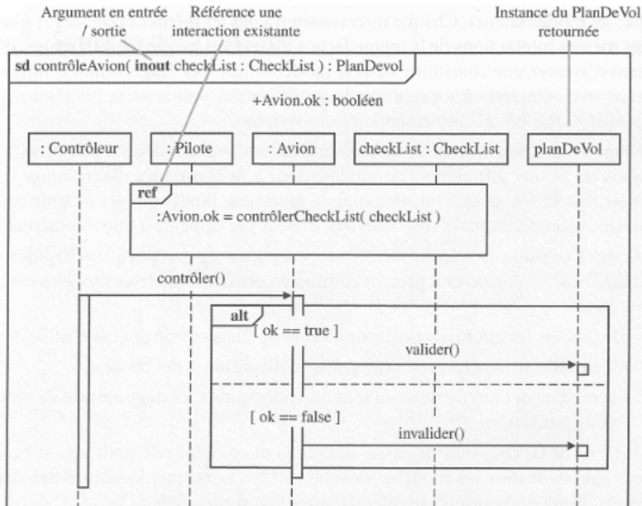
ref

Contrôler check list

ref

Procédure de décollage

Exemple : décollage d'avion



Exemple : caisse

Caissier

: Caisse

Client

LOOP

Saisir numéro article

Afficher libelle et prix de produit

Afficher libelle et prix de produit

Afficher Total

Afficher Total

Génie Logiciel
- une introduction

Cours 5 : UML, diagrammes de classe et diagrammes d'état/transition

Vue globale des diagrammes UML

Diagramme

Diagramme Structurel

Diagramme Comportemental

Diagramme de Classes

Diagramme de Composants

Diagramme d'Objets

Diagramme d'Activités

✔ Diagramme de Cas d'Utilisation

Diagramme d'Etats

Diagramme de Structure Composite

Diagramme de Déploiement

Diagramme de Packages

Diagramme d'Interaction

✔ Diagramme de Séquence

Diagramme de Vue Globale d'Interaction

Diagramme de Communication

Diagramme de Timing

Diagrammes de classe

- Description de l’organisation d’un système
  - interne = entités + relations entre entités
  - statique = pas de comportement, cycle de vie d’une entité, etc.
- Utilisation :
  - Spécification des fonctionnalités : interfaces assurées
  - Conception : capture des concepts et de leur relation
    - Voir cours BD
  - Implémentation : contenu et implémentation d chaque classe

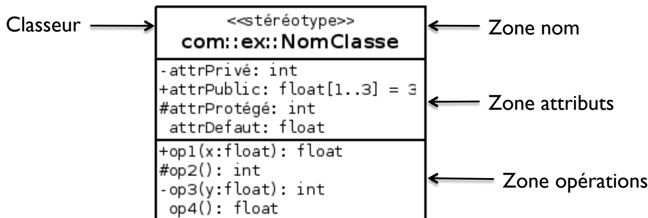
Eléments des diagrammes de classes

- Classes
  - Déclaration de nom :

    [**<<**stéréotype**>>**] [Paquetage1**::** … **::** PaquetageN **::**]

    NomClasse [**{** [**abstract**,] [auteur,] [état,] … **}**]

    - Stéréotypes : abstract, interface, enumeration, utility, …



Eléments des diagrammes de classes

- Classes
  - Attributs = informations stockées dans la classe

    ```
    [visibilité] [/] nom:type [ [multiplicité] ] [= valInit]
    ```

    - Visibilité : privé (-), public (+), protégé(#), défaut (paqt)
    - Primitif ou dérivé(/)

  - Opérations (déclaration de méthode)

    ```
    [visibilité] nom([paramètres]):typeRetour [{ propriété }]
    ```

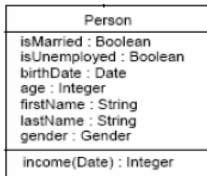
    - Propriété en OCL
      - = logique du 1er ordre avec macro-définitions
      - Pre/post conditions, invariants, relations, …

Exemple de propriétés OCL

- Attribut

```
{ inv: self.age > 0 }
{ init: 0 }
```

(self≅this)



- Opération

```
+income(d:Date) : Integer
  { post: result = age * 100 }
+getCurrentSpouse() : Person
  { pre: self.isMarried = true
    body: self.mariages->select(m|m.ended = false).spouse }
```

- Classe

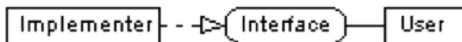
```
{ inv: let income:Integer = self.job.salary->sum() in
  if isUnemployed then income < 100
  else income >= 100
  endif }
```

- Relation

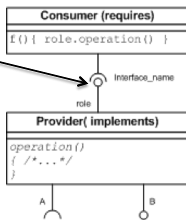
```
{ordered} {subset} {or}
```

Eléments des diagrammes de classes

- Interface = ensemble d'opérations qu'une classe doit implémenter
  - Stéréotype spécial de classeur de classe
  - Deux notations graphiques
    - Classeur avec angles arrondis

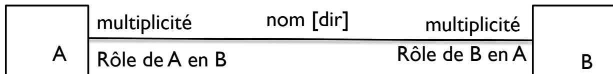


    - Balle (ou « sucette »)
      - Ex: A interface requise,
      - Ex: B interface fournie



Eléments des diagrammes de classes

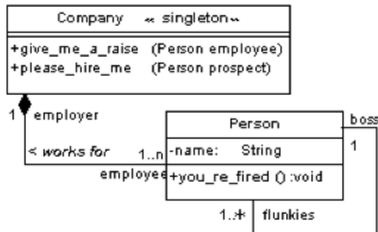
- Relation d'association entre classes



  - multiplicité =
    - 1 (unique) – par défaut
    - n (inconnue mais bornée)
    - k .. l (constante ou intervalle)
    - k..* (k ou plus)
  - dir = direction de lecture de la relation (> ou <)
  - une relation peut-être contrainte avec OCL, ex `{ordered}`
  - les relations peuvent être n-aire (n ≥ 2)
    - non recommandé car trop complexe
    - utiliser les classes-association (voir p. 10)

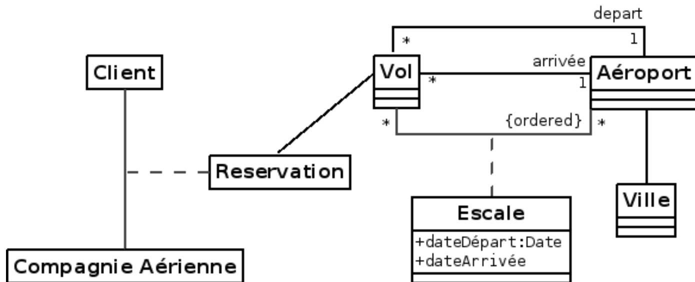
Exemple

- Lecture de diagramme :
  - Company
    - classe ayant une seule instance
    - avec deux opérations publiques
  - Person
    - classe avec attribut privé nom
    - une opération publique
  - Relations
    - 1..n Person « works for » 1 Company
    - Person a le rôle d'employé (plusieurs employées pour une Company)
    - Company a le rôle de employeur (un employeur pour une Person)
    - Person a le rôle de boss (un par personne)
    - Person a le rôle de subalterne (plusieurs pour un boss)



Eléments des diagrammes de classes

- Cas particuliers d'association :
  - Classe-association
    - = relation exprimée dans une classe
    - Exemple : réservation et vols d'une compagnie aérienne



Eléments des diagrammes de classes

- Cas particuliers d'association :
  - Agrégation

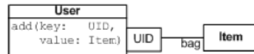
    = détruire le contenant ne détruit pas le contenu

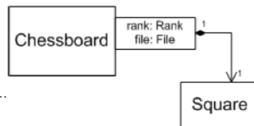
  - Composition

    = détruire le contenant détruit le contenu

  - Composition quantifiée

    = indexation selon un domaine





Exemple

Bank

accountNumber:Integer

0..1

0..* customer

«enumeration»
Gender

male
female

Person

isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
gender : Gender

income(Date) : Integer

manager

1

0..*

managedCompanies

Company

name : String
numberOfEmployees : Integer

stockPrice() : Real

employee

0..*

employer

0..*

wife

0..1

husband

0..1

Job

title : String
startDate : Date
salary : Integer

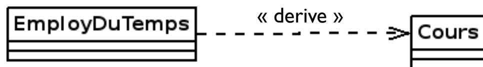
Marriage

place : String
date : Date

Eléments des diagrammes de classes

- Relations de dépendance

  - Dépendance sémantique, la modification de la cible (B) implique la modification de la source (A)
  - Stéréotype : (optionnel)
    - « friend » : visibilité spéciale à la classe source (A) sur la classe cible (B), voir C++
    - « derive » : A calculée à partir de B
    - « call » : opération en A appelle opération en B
    - « create » : A crée une instance de B
    - …



Eléments des diagrammes de classes

- Relation d'héritage
  - Outil pour l'abstraction
  - Concept de la programmation objet
  - Propriétés :
    - Classe enfant (A) possède les propriétés de ses classes parents (B)
    - L'enfant A peut redéfinir les méthodes du parent B
    - Toute association du parent (B) s'applique par défaut à ses enfants (A)
    - La classe enfant (A) peut être utilisée à la place de la classe parent (B)
    - Une classe enfant (A) peut avoir plusieurs parents (B)
      - Mais l'héritage multiple n'est pas recommandé/présent dans tous les langages

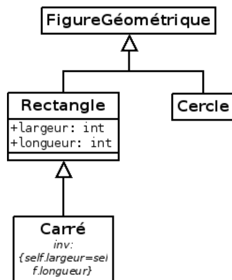


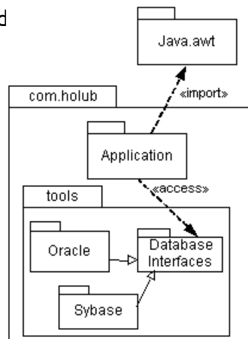
Diagramme de classe : démarche

Etapes à suivre pour la définition d'un diagramme de classe :

1. Trouver les classes du domaine
   - identifier les concepts (substantifs du domaine)
   - ne pas inclure des concepts redondants
   - ne pas donner des détails d'implémentation
2. Trouver les associations entre les classes
   - éviter les associations qui impliquent plus de 2 classes
   - éviter les chemins multiples entre classes qui limitent leur réutilisation
3. Trouver les attributs des classes
   - Complètements d'information sur les classes
4. Organiser et simplifier le diagramme en utilisant l'héritage
   - Voir aussi les patterns de conception classiques
5. Tester les chemins d'accès aux classes
6. Itérer à partir de 1, affiner le modèle
   - éliminez les classes et les relations redondantes
   - décomposez les classes ayant trop de responsabilités

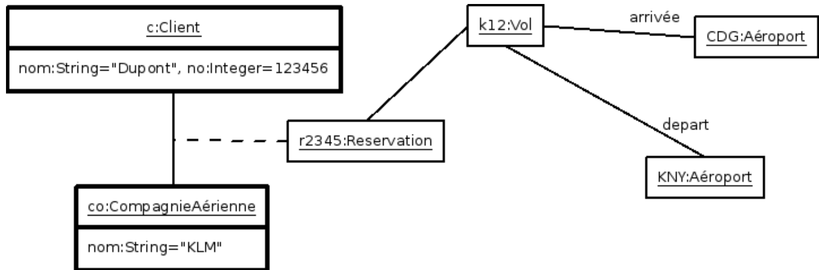
Diagrammes de paquetage

- Spécifient l'organisation des classes
  - Indispensable pour les grands modèles
  - Difficile à définir exactement dès le début, des itérations sont nécessaires à chaque révision d produit.
- L'unité de description est le paquetage
  - = groupement de classes/sous-paquetages ayant des fonctionnalités similaires
- Relations entre paquetages :
  - Imbrication
  - Dépendance (typée)
  - Héritage



Diagrammes d'objets

- Décrivent les objets du système et leur relation à un instant donné (*snapshot*).
- Vue dynamique du diagramme de classes.
- Exemple : compagnie aérienne



Diagrammes de composants

- Décrivent l'organisation dynamique d'un système à un instant donné (*snapshot*)
- L'unité de base est le composant
  - = unité autonome constituée d'un ensemble d'objets qui collaborent pour offrir un ensemble de fonctionnalités via une interface

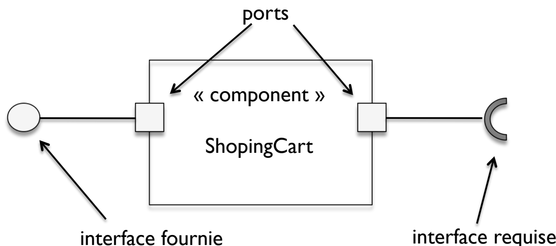
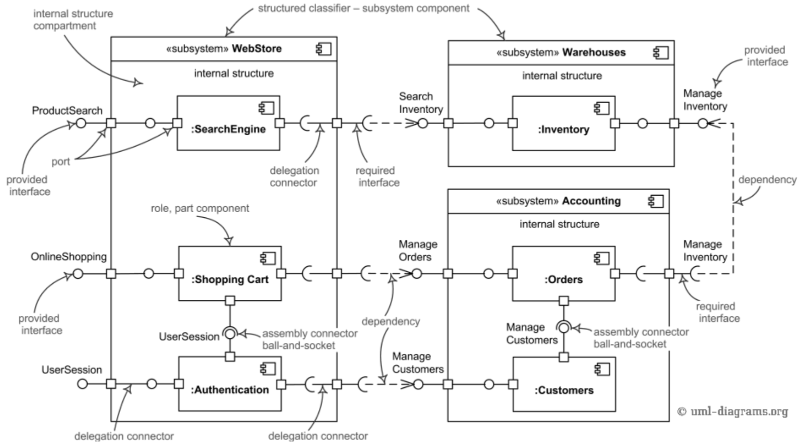


Diagramme de composants

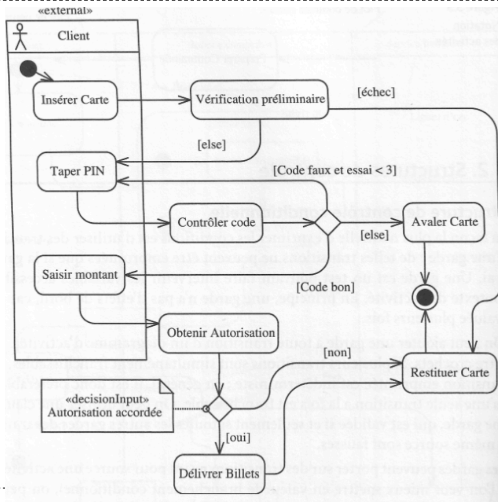


Diagrammes d’activités

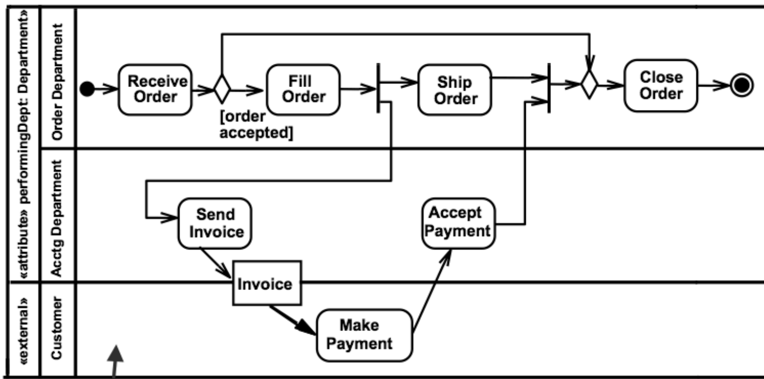
- Définit le flot de contrôle d’un système ou d’une opération
- Eléments :
  - Activité = traitement
    - Le contrôle reste dans l’activité tant qu’elle n’est pas finie
    - Peuvent être imbriquées hiérarchiquement = activité composite
  - Transition = changement d’activité
    - Déclenchée dès que les activités en entrées sont terminées
    - Sans durée, atomiques
    - Servent à créer (fork) et a synchroniser (join) les activités

- Modèle théorique = réseaux de Petri

Exemple : terminal bancaire



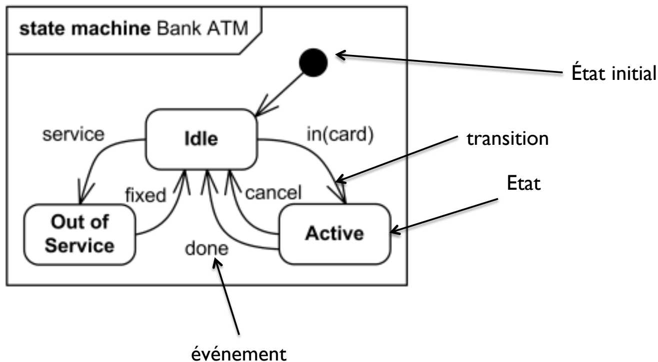
Exemple : système de facturation



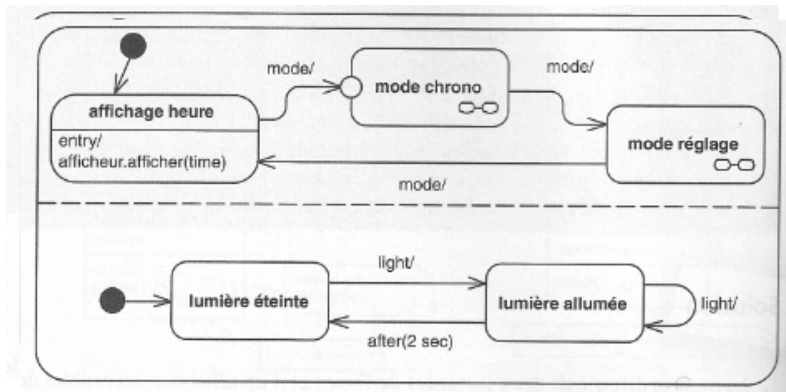
Diagrammes d'état/transition

- *Statecharts*
- Décrivent le comportement interne à l'aide d'automates d'états finis
  - Etat = caractérise le passé de l'automate
    - Peuvent être « composites » = à l'intérieur de l'état est définie une machine
  - Transition = changement d'état, caractérise les actions/ réactions de l'automate
    - Déclenché par des événements (appel de méthode, changement e variable, réception d'un signal)
    - Relation binaire entre états
    - + Syntaxe spécifique pour alléger les diagrammes

Exemple



Exemple : montre digitale



Exemple : montre digitale