

Génie Logiciel - UML

Franck Ledoux
LaMI – Université d'Evry Val d'Esonne
fledoux@lami.univ-evry.fr

Organisation du cours

- 13 séances de 3 heures
- chaque séance : cours + TDs

Contrôle des connaissances :

- 1 examen de 3 heures (dernier séance)
- pas de contrôle continu
- 1 projet au second semestre

Ressources intéressantes

- **Livres en français**
 - UML – Modéliser un site e-commerce
P. Roques, *Eyrolles*
 - Le guide de l'utilisateur UML
G. Booch, J. Rumbaugh, I. Jacobson, *Eyrolles*
 - Modélisation objet avec UML
P. A. Muller, *Eyrolles*
 - Précis de génie logiciel
M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot, *Masson*
- **Sites Web**
 - www.uml.free
 - ...

Objectif de l'enseignement

- Analyse et conception de système informatique
- Phase amont de l'activité de programmation
- Utilisation de la notation UML



Assimiler l'importance des activités de spécification, d'analyse et de conception par rapport à l'activité de programmation

Pourquoi un cours GL ?

- Vos compétences : « programming-in-the-small »
 - Programmation **individuelle** sur de **petits** problèmes
 - Algo, langages de programmation, structures de données
 - (parfois) un peu de méthodologie : analyse descendante
- En entreprise : « programming-in-the-large »
 - Travail en **équipe** sur des projets **longs et complexes**
 - Spécifications de départ peu précises
 - Dialogue avec le client/utilisateur : parler métier et non informatique
 - Organisation, planification, gestion du risque



Démarche ingénierique : génie logiciel

Planning du cours

- **Séance 1** - Introduction au GL, approche OO et UML-RUP
- **Séance 2** - Cas d'utilisation : cours + TD
- **Séance 3** - Diagramme de séquence : cours + TD
- **Séance 4** - Diagramme d'interaction : cours + TD
- **Séance 5** - Diagramme de classe : cours + TD
- **Séance 6** - Diagramme d'états-transitions : cours + TD
- **Séance 7** - Diagramme de composant et de déploiement : cours + TD
- **Séance 8** - ??
- **Séance 9** - RUP + cas d'étude 1 (1/2) : cours + TD
- **Séance 10** - cas d'étude 1 (2/2) : TD
- **Séance 11** - cas d'étude 2 (2/2) : TD
- **Séance 12** - cas d'étude 2 (2/2) : TD
- **Séance 13** - **Examen**

d'ici fin 2003
février 2004

Introduction au Génie Logiciel

Le Génie Logiciel

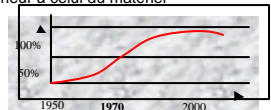
- Le terme **Génie Logiciel** est né entre le 7 et le 11 octobre 1968 à Garmish-Partenkirchen sous le nom de **software engineering** sous le parrainage de l'OTAN
- Défini par un groupe de scientifiques pour répondre à un problème bien défini s'énonçant en 2 constatations :
 - le logiciel n'était pas **fiable**
 - il était incroyablement difficile de réaliser dans des **délais prévus** des logiciels satisfaisant leur **cahier des charges**

Le Génie Logiciel

- **Une définition**
Spécifier, concevoir, construire, maintenir de grands systèmes logiciels

Méthodologie de construction en **équipe** d'un logiciel complexe et à **multiples versions**
 - **Programmation vs Génie Logiciel** (approximation)
 - Programmation : activité personnelle
 - Génie Logiciel : activité d'équipe
- Suivant les projets, la partie programmation (codage) ne représentera qu'entre 10% et 30% du coût total.

Logiciel : aspects économiques

- Importance économique du logiciel
 - importance croissante de l'informatique dans l'économie (1985 : 150 Mrd\$ - 1995 : 450 Mrd\$)
 - coût du logiciel supérieur à celui du matériel
- 
- L'économie de toute nation industrialisée est dépendante du logiciel. La dépense en logiciel représente une partie significative de son PNB
 - coût maintenance supérieur au coût de conception
- ➔ **améliorer la qualité du logiciel !**

Erreurs célèbres et projets douloureux (qui justifient l'utilisation de GL)

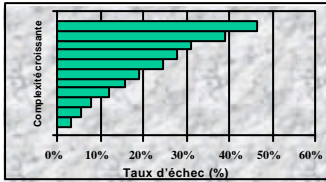
- 1971 : lors d'une expérience météorologique en France, 72 ballons sondes détruits à cause d'un défaut logiciel
- La sonde Mariner vers Vénus perdue dans l'espace à cause d'une erreur dans un programme (virgule remplacée par un point)
- 1981, le premier lancement orbital de la navette spatiale retardé de 2 jours à cause d'un problème logiciel. Elle fut lancée sans que l'on ait exactement localisé la cause du problème
- du 15 au 16 décembre 1990, les abonnés de ATT sur la côte est des Etats-Unis furent privés de tout appel longue distance à cause d'une réaction en chaîne dans le logiciel du réseau due à un changement de version du logiciel.

Erreurs célèbres et projets douloureux

- Avion F16 retourné au passage de l'équateur : non prise en compte du référentiel hémisphère Sud
- OS-360 d'IBM (années 1960) a été livré en retard, a nécessité plus de mémoire que prévu, son prix de revient dépassant de beaucoup les estimations, et ses premières versions comportant des erreurs.
- Compilateur PL1 chez Control Data jamais abouti (années 1970)
- abandon du projet d'informatisation de la bourse de Londres : 4 années de travail et 100 M£ perdus
- Abandon du système de trafic aérien américain
- Retard (1 an) du système de livraison des bagages de l'aéroport de Denver
- Bogue de l'an 2000, instabilité de Windows 95 ...

Enjeux du logiciel

- Nécessité de méthodes, de processus pour le développement des logiciels coûteux et complexes
- Primordial pour les **systèmes critiques** : nucléaire, transport, systèmes bancaires



➡ améliorer la qualité des logiciels

Complexité croissante du logiciel

- système offrant de plus en plus de **fonctionnalités**
- systèmes distribués** : machines hétérogènes en réseau
- mutations technologiques** rapides : langages et environnements de développement, O.S., matériel.
- évolution des besoins du client** en cours de projet

Qualité du logiciel

- Privilégier la qualité à l'efficacité
 - la prévention des erreurs coûte des dizaines de fois moins cher que leur correction
 - démarche qualité : ISO 9126 (www.osil.ch/eval/node15.html)
 - gérer la complexité des logiciels tout au long de leur cycle de vie
 - séparer les aspects fonctionnels et technologiques
 - décomposition en sous-systèmes
 - démarche itérative
- Qualité externe vs. Qualité interne
 - externe : vision client
 - interne : vision du développeur

} approche objet

Facteurs de qualité du logiciel

- Qualité externe
 - complétude fonctionnelle** : réalise les tâches attendues
 - maniableté** : facilité d'utilisation
 - Interface utilisateur appropriée
 - Documentation complète et précise
 - fiabilité** :
 - fonctionne même dans des cas atypiques
 - il ne doit pas causer de dommages physiques ou économiques en cas de défaillance
 - adaptabilité** : adaptation aux modifications
- Qualité interne
 - réutilisation** : il doit être possible de faire évoluer le logiciel pour répondre à de nouveaux besoins
 - traçabilité** : suivi précis de l'analyse à l'implantation
 - efficacité** : bonne utilisation des ressources matérielles
 - portabilité** : adaptation à de nouveaux environnements

Génie logiciel : En résumé

- Le GL doit fournir des **méthodes de conception** de systèmes complexes permettant :
 - une prise en compte du client
 - une démarche qualité
 - une organisation du travail en équipe
 - une meilleure évolution et maintenance du logiciel
 - un respect des coûts et délais estimés initialement
- Le GL, ce sont aussi des outils associés :
 - ateliers de génie logiciel
 - méthodes de développement

Exemple : Rational Rose (UML)

Modèles de développement du logiciel

Modèles / Processus

- Une définition

Procédé de production établi comme une **suite de descriptions** de plus en plus précises et de plus en plus proches d'un programme exécutable et de sa documentation

- Notion de **raffinements successifs** (passage d'une description à une autre)
- **Nature itérative**, certaines étapes déclenchent la révision du résultat des étapes précédentes

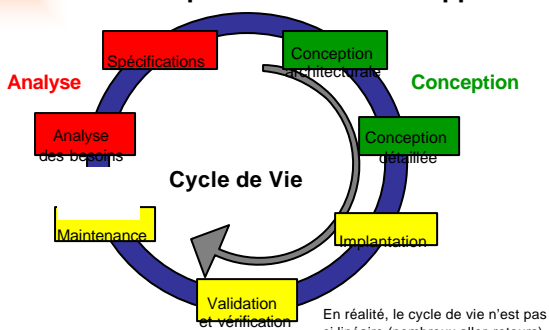
- Large place pour les phases d'analyse, de conception et de validation

- **But** : obtenir un processus **rationnel, reproductible et contrôlable**

Caractéristiques d'un processus

- **Compréhensible** : clairement défini et compréhensible
- **Observable** : l'évolution du produit est visible de l'extérieur
- **Accepté** par les acteurs du projet
- **Sûre** : les erreurs sont détectées avant la mise en service du produit
- **Robuste** : les problèmes non prévus ne stoppent pas la réalisation du produit
- **Maintenable** : le processus évolue en fonction des besoins de changements d'organisation
- **Efficace** : le temps de réalisation du produit

Activités du processus de développement



Défis des processus logiciels

- Généralement, les spécifications sont incomplètes et anormales
- La frontière entre spécification, conception et réalisation est floue
- Les tests ne sont pas réalisés dans l'environnement définitif du système
- Un logiciel ne connaît pas l'usure - maintenir ne veut pas dire remplacer un composant

Analyse des besoins

- **But** : éviter de développer un logiciel non adéquat
- Étude du domaine d'application, de l'environnement du futur système afin d'en déterminer le rôle, les frontières...
- **Dialogue avec le client** qui fournit les données du problème
- **Pas de discussion techniques**, on cerne les besoins : entretiens, questionnaires, étude de situation similaire...
- **Résultat** :
 - documents décrivant les aspects pertinents de l'environnement du futur système, son rôle et sa future utilisation **cahier des charges**
 - Partage logiciel/matériel déterminé suivant des facteurs qualités : robustesse, efficacité, portabilité...

NB : activité essentielle au début du processus, elle est couplée avec les **études de faisabilité** et la **planification**. Elle se poursuit durant tout le cycle de vie du logiciel (questions relatives aux besoins et à l'environnement peuvent émerger).

Spécifications

- **But** : établir une première description du futur système
- Document précis spécifiant les fonctionnalités attendues, rédigés **formellement** (spécifications algébriques,...) ou **semi-formellement** (UML,...)
- **Résultat** : une description de **ce que doit faire** le système (et pas comment) compréhensible par le client/utilisateur
- Fortement corrélée avec l'analyse des besoins et la validation

NB :

- Une première version du manuel de référence est parfois produite à cette étape
- **Les spécifications ne sont jamais complètes et définitives** (évolution du domaine, besoins supplémentaires)



Conception

- **But : enrichir la description du logiciel de détails d'implantation afin d'aboutir à une description très proche d'un programme**
- Conception architecturale :
 - décomposer le logiciel en **composants** plus simples.
 - Préciser les **interfaces** et **fonctionnalités** de chaque composant
 - **Résultat** : description de l'architecture logicielle et spécifications de ses composants
- Conception détaillée :
 - Pour chaque composant, on indique **comment** sont réalisées ses fonctionnalités : représentation des données, algorithmes
- Expertise informatique : hors compréhension du client
- Frontière floue entre spécifications et conception :
 - La conception commence souvent pendant la spécification (contraintes de réalisation à anticiper)
 - Elle peut remettre en cause la spécification (aller-retours)



Implantation

- Souvent **trop de temps consacré au codage** au détriment des phases d'analyse et de conception : mauvaise pratique parfois très coûteuse...
- Dans un projet bien conduit, l'effort se décompose environ comme suit :
 - 40% pour la spécification et la conception
 - 20% pour l'implantation
 - 40% pour la validation et la vérification
- Activité la mieux **maîtrisée**, « outillée », voire **automatisée**
- Savoir user de la **réutilisabilité des composants**, voire d'outils de génération de code (mise en place automatique du squelette du code à partir du modèle système)



Correction – la validation

- La **validation** répond à la question : a-t-on décrit le « bon » système ?
- **Difficulté** : l'imprécision des besoins et des caractéristiques du système à développer
- Elle consiste en des **revues** et **inspections** de spécifications ou de manuels et du **prototypage rapide**
- **Maquettage** ou prototypage rapide : développement rapide d'un programme qui est une **ébauche** du futur système
- Soumis à des scénarios d'utilisation, il permet de préciser les souhaits du client



Maquette exploratoire



Maquette expérimentale

- Lors d'une étape de conception, plusieurs maquettes peuvent être comparées pour valider différents choix



Correction – la vérification (1/2)

- La **vérification** répond à la question : le développement est-il correct par rapport à la spécification initiale ?
- Elle inclut les activités de **preuve** et de **test**
- Une **preuve** porte sur une spécification détaillée ou un programme et permet de prouver qu'il ou elle satisfait bien la spécification de départ
- Le **test** recherche des erreurs dans une spécification ou un programme



Correction – la vérification (1/2)

- **Test statique** : examen ou analyse du texte
- **Test dynamique** : exécution sur un sous-ensemble fini de données possibles
- Selon l'avancement du développement, on distingue plusieurs types de test :
 - Le **test unitaire** consiste à tester des composants isolément
 - Le **test d'intégration** consiste à tester un ensemble de composants qui viennent d'être assemblés
 - Le **test système** consiste à tester le système sur son futur site d'exploitation dans des conditions opérationnelles et au-delà (surcharge, défaillance matérielle,...)
- Testeur ¹ concepteur du programme
- Activités souvent sous-estimées



Maintenance

- Deux types de maintenance
 - Correction des **erreurs du système**
 - Demande d'**évolution** (modification de l'environnement technique, nouvelle fonctionnalités,...)
- Facteurs de qualités essentiels
 - Corrections : **robustesse**
 - Évolutions : **modifiabilité, portabilité**
- Étape longue, critique et coûteuse
 - 80% de l'effort de certaines entreprises (pb de pratiques ?)

Cycles de vie « classiques »

Basés sur la **succession** des différentes étapes processus de développement

- Cycles de vie linéaires
 - Modèle en **cascade**
 - Modèle en « **V** »
- Cycle de vie itératif
 - Modèle en **spirale** (ou incrémental) : prototypes

Modèle en cascade

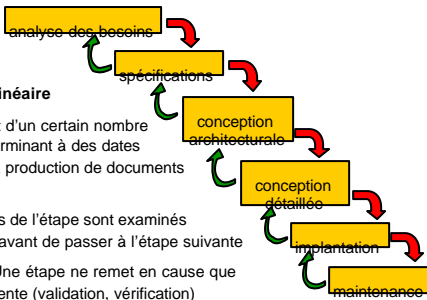
Principes

- Processus **linéaire**
- On convient d'un certain nombre d'étapes, se terminant à des dates précises par la production de documents ou logiciels
- Les résultats de l'étape sont examinés attentivement avant de passer à l'étape suivante
- **Itération** : Une étape ne remet en cause que l'étape précédente (validation, vérification)

Modèle en cascade

Principes

- Processus **linéaire**
- On convient d'un certain nombre d'étapes, se terminant à des dates précises par la production de documents ou logiciels
- Les résultats de l'étape sont examinés attentivement avant de passer à l'étape suivante
- **Itération** : Une étape ne remet en cause que l'étape précédente (validation, vérification)



Modèle en cascade : inconvénient

- Validation limitée à un pas d'itération
 - augmentation des risques : erreur d'analyse ou de conception très coûteuse si détectée trop tard !
- Difficile d'effectuer des modifications en cours de route
- Solution limitée aux petits projets
 - Bien adapté lorsque les besoins sont **clairement identifiés et stables** i.e. les risques sont bien délimités dès le début du projet
 - projet **court** avec **peu de participants**

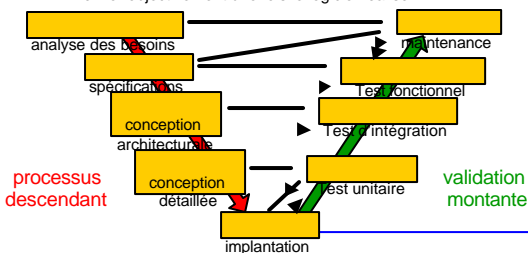


Souvent abandonné au profit du modèle en « V », plus récent, qui présente une articulation plus réaliste entre l'activité de réalisation et celle de validation-vérification

Modèle en « V »

Principe : expliciter le fait que les premières étapes de développement (analyse) préparent les dernières étapes (correction)

Prémunit d'énoncer une propriété qu'il serait impossible de vérifier objectivement une fois le logiciel réalisé



Modèle en « V » : intérêts

Validations intermédiaires

- Prévention des erreurs : validations des produits à chaque sortie d'étape descendante
- Préparation des protocoles de validation finaux à chaque étape descendante
- Validation finale montante : confirmation de la pertinence de l'analyse descendante
- Limitation des risques en cascade par validation de chaque étape
- Existence d'outils support

- **Modèle éprouvé très utilisé pour de grand projets**

Mais...

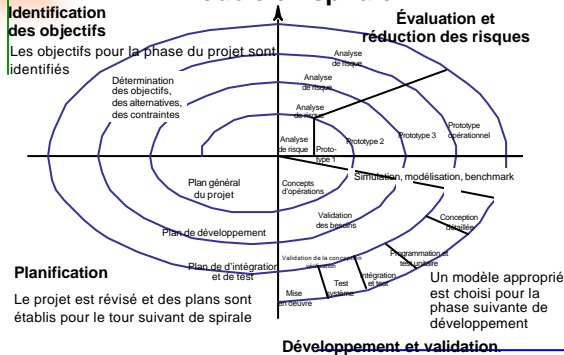
Modèle en « V » : limitations

- **Un modèle toujours séquentiel...**
 - Prédominance de la documentation sur l'intégration : validation tardive du système par lui-même
 - Les validations intermédiaires n'empêchent pas la transmission des insuffisances des étapes précédentes
 - Manque d'adaptabilité
 - Maintenance non intégrée : syndrome du logiciel jetable
- **... adapté aux problèmes sans zones d'ombres**
 - Idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires

Modèle en spirale

- **Modèle itératif basé sur le prototypage**
 - *Idee* : fournir le plus rapidement possible un **prototype** exécutable permettant une validation concrète et non sur document
 - accent mis sur l'**analyse de risque**
 - Progression du projet par incréments successifs de versions successives du prototype : **itérations**
 - 1 itération = 1 mini-cycle de vie en cascade
 - Certains prototypes peuvent être montrés aux clients et utilisateurs. Par ailleurs, une **maquette** peut être réalisée préalablement au premier prototype (Prolog)
 - La validation par prototypes ne justifie pas l'absence de recours à la **documentation**

Modèle en spirale



Formulaire d'un tour de spirale

- Objectifs
- Contraintes
- Alternatives
- Risques
- Résolution des risques
- Résultats
- Plans
- Implication

Exemple : amélioration de la qualité

- Objectifs
 - Amélioration significative de la qualité
- Contraintes
 - Sur une échelle de trois ans
 - Sans gros investissements
 - Sans changement des standards actuels
- Alternatives
 - Réutiliser des logiciels certifiés existants
 - Introduire des spécifications et des vérifications formelles
 - Investir dans des outils de test et de vérification

Exemple : amélioration de la qualité

- Risques
 - Pas d'amélioration rentable de la qualité
 - L'amélioration risque de gonfler les coûts
 - Les nouvelles méthodes peuvent pousser le personnel à partir
- Résolution des risques
 - Se documenter
 - Faire un projet pilote
 - Étudier les composants réutilisables potentiels
 - Évaluer les outils de support disponible
 - Former et motiver le personnel



Exemple : amélioration de la qualité

- Résultats
 - Le manque d'expérience en méthodes formelles rend l'amélioration difficile
 - On dispose d'outils limités pour le développement standard de la société
 - On dispose de composants réutilisables, mais peu d'outils de support pour la réutilisation
- Plans
 - Explorer l'option réutilisation plus en détail
 - Prototyper des outils de support à la réutilisation
 - Etudier les traités de certification de composants
- Implication
 - Financer 1 mois complémentaires de développement



Exemple : catalogue de composants

- Objectifs
 - Fournir un catalogue de composants logiciels
- Contraintes
 - En une année
 - Doit gérer les types de composants existants
 - Coût total inférieur à \$100, 000
- Alternatives
 - Acheter un logiciel de recherche d'information
 - Acheter une base de données et développer un catalogue
 - Développer un catalogue dédié



Exemple : catalogue de composants

- Risques
 - Peut être impossible au vue des contraintes
 - Les fonctions du catalogue peuvent être inappropriées
- Résolution des risques
 - Développer un catalogue prototype pour clarifier les besoins
 - Lancer une consultation sur les systèmes existants
 - Relâcher les contraintes de temps



Exemple : catalogue de composants

- Résultats
 - Les systèmes de recherche d'information ne sont pas flexibles. On ne peut pas répondre aux besoins
 - On peut enrichir un prototype basé sur un SGBD pour compléter le système
 - Le développement d'un catalogue dédié n'est pas rentable
- Plans
 - Enrichir un prototype basé sur un SGBD et améliorer l'interface utilisateur
- Implication
 - Financer les 12 mois prochains de développement



Modèle en spirale : intérêts

- Validation concrète et non sur documents
- Limitation du risque à chaque itération
- Concentre l'attention sur les options de réutilisation
- Client partenaire : retour rapide sur attentes
- Progressivité : pas d'explosion des besoins à l'approche de la livraison (pas de « *n'importe quoi pourvu que cela passe* »)
- Flexibilité :
 - Modification des spécifications = nouvelle itération
 - Maintenance = forme d'itération
- Planification renforcée



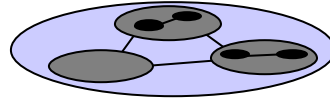
Modèle en spirale : limitations

- Ce n'est pas le processus parfait
 - Cycle itératif : planification très attentive et rigoureuse, mais peut dérouter au premier abord
 - Cycle en « V » : processus éprouvé le plus répandu, surtout pour les systèmes connus.
- Nécessite des expertises sur l'évaluation des risques
- Processus adapté à la modélisation objet
 - Modèle objet : se prête parfaitement à une démarche incrémentale

Approche orientée objet

Quelle architecture logicielle

- Développement logiciel = décomposer / réunir
 - Diviser pour comprendre : décomposer le système en sous-parties pour maîtriser la complexité
 - Réunir pour construire : assurer la cohérence globale
- Architecture logicielle : comment décomposer ?



De l'approche fonctionnelle vers...

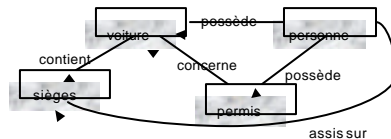
- **Décomposition fonctionnelle**
 - Initialement les informaticiens ont raisonné en terme de fonctions du système
 - Méthodes inspirées directement de l'architecture des ordinateurs
 - Décomposition **hiérarchique** suivant des critères fonctionnels



- **Limitations**
 - Hiérarchie fonctionnelle : **remise en cause difficile**
 - **Adaptabilité limitée** et coûts des erreurs important

... vers l'approche orientée objet

- Décomposition « systémique »
 - Ensemble hétérarchique de **composants** (les objets) **indépendants** (encapsulation) dont la **collaboration dynamique** fonde les fonctionnalités du système.
 - Objet défini comme une abstraction du monde réel



L'approche OO : avantages

- **Stabilité** dans la modélisation par rapport aux entités du monde réel
- Adéquation avec un cycle **itératif** de développement
- **Équilibre** traitement / données
- Possibilité de **réutiliser / porter** des éléments d'un autre développement
- **Simplicité** du modèle – 5 concepts :
 - Objets, messages, classes, héritage et polymorphisme

L'approche OO : avantages

- Développer des logiciels fondés
 - Sur la modélisation des objets du monde réel
 - Sur l'emploi de ce modèle pour bâtir une conception indépendante de tout langage organisé autour de ces objets (C++, Eiffel, Java, SmallTalk,...)
- Meilleure compréhension des besoins
- Conception plus **propre**
- Système plus facile à maintenir

Concepts objets

- Objets
 - État
 - Comportement
 - Identité
- Relations entre objets
 - Message
 - Interface
 - abstraction
- Classe / instance
- Héritage
- Polymorphisme

Objet

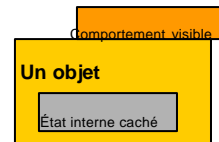
- Un objet est une **entité** du monde réel ou du monde informatique
- Exemples d'objets
 - Un port, un bateau, un tonneau
 - Une fenêtre, un menu, une icône
 - Un entier, un réel, une chaîne de caractères
- Qu'est-ce qui n'est pas un objet ?
 - « Tout est objet » [Ferber, 1983]
 - Généralement, dans les langages de programmation, on fait le choix du « tout objet » sauf parfois, les types de base (entier, réel, caractère, booléen) pour des raisons d'efficacité

Objet (concrètement)

- Unité **atomique** formée de l'union d'un **état**, d'un **comportement** et d'une **identité**
 - État : décrit les propriétés de l'objet à un moment donné
 - Comportement : définit les propriétés dynamiques de l'objet
 - Comment il agit et réagit aux informations qui lui parviennent de son environnement
 - Identité : caractérise de manière univoque l'existence propre de l'objet (clé en BD)
- L'objet fournit une relation d'**encapsulation** qui assure à la fois une cohésion interne très forte et un faible couplage avec l'extérieur

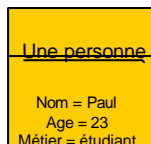
Objet

- Chaque objet contient un état interne qui lui est propre et un comportement accessible aux autres objets



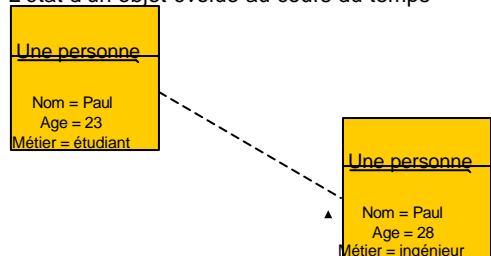
Objet : état

- État = ensemble d'**attributs**
 - Chaque attribut caractérise une **propriété précise**
 - Les attributs peuvent prendre à un moment donné toute valeur d'un domaine de définition donné
- L'état regroupe les valeurs **instantanées** de tous les attributs d'un objet



Objet : état

- L'état d'un objet évolue au cours du temps



Objet : comportement

- Le comportement regroupe toutes les compétences d'un objet et décrit ses actions et réactions via la notion de **méthodes**
- Comportement = ensemble de **méthodes**
 - Méthode = opération atomique qu'un objet réalise soit de son propre chef, soit en réaction à une stimulation de l'environnement (envoi de message d'un autre objet)
 - L'action réalisée **dépend de l'état** de l'objet

Une personne
Nom = Paul Age = 28 Métier = ingénieur
ChangerAge ChangerMétier

Franck Ledoux DESS Bio-Informatique 2003-2004 61

Objet : comportement

- Types de méthodes
 - constructeur** : crée et initialise l'objet → CréerPersonne
 - modificateur** : modifie l'état de l'objet → ChangerAge
 - observateur** : donne une information sur l'état → DonnerAge
 - destructeur** : détruit l'objet → OterPersonne
- Cycle de vie d'un objet**
Tout objet est créé, évolue et est détruit...

Franck Ledoux DESS Bio-Informatique 2003-2004 62

Objet : identité

- Existence propre d'un objet
 - Permet de distinguer des objets ayant le même état
 - Gérée **implicitement** : pas d'attribut correspondant
 - Si on le souhaite, on peut cependant rajouter un identifiant dans l'état de l'objet (clé naturelle) :
 - Numéro INSEE
 - Numéro étudiant

Notion de clé primaire en base de données

- Le concept d'identité reste **indépendant** du concept d'état

Franck Ledoux DESS Bio-Informatique 2003-2004 63

Relations entre objets : messages

- Système = **ensemble d'objets en relation**
 - Dynamique du système : **collaboration** entre objets
 - Interaction non structurée par **passage de messages**
 - La communication entre objets est la grande différence entre l'approche fonctionnelle et l'approche objet
- Le concept de message
 - Le message est le support d'une relation de communication qui relie, de façon dynamique, les objets qui ont été séparés par le processus de décomposition
- La notion de message est un concept abstrait mis en œuvre :
 - Appel de procédure, événement discret, interruption,...

Franck Ledoux DESS Bio-Informatique 2003-2004 64

Relations entre objets : messages

- Interface d'un objet** : contrôle de la modularité
 - Interface = méthodes invocables par l'extérieur
 - L'état de l'objet ne peut être modifié par l'extérieur qu'en invoquant une méthode de l'interface
 - L'objet peut refuser de répondre à une invocation externe

L'objet garde toujours la maîtrise de son état

- Abstraction de données
 - Un objet est complètement défini par son interface
 - Les autres objets n'ont pas à connaître l'implémentation interne de l'objet pour communiquer avec lui

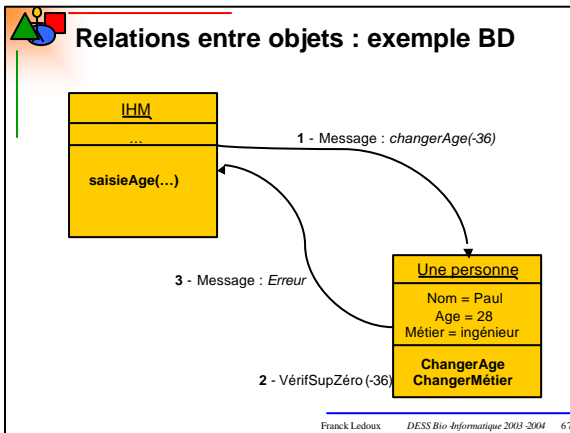
Franck Ledoux DESS Bio-Informatique 2003-2004 65

Relations entre objets : exemple BD

- Méthodes publiques : interface
 - changerNom; changerAge, changerMétier
 - DonnerNom, DonnerAge, DonnerMétier
- Méthodes d'implantation
 - VérifMajuscule, VérifSupZéro

Une personne
Nom = Paul Age = 28 Métier = ingénieur
ChangerAge ChangerMétier

Franck Ledoux DESS Bio-Informatique 2003-2004 66



Relations entre objets : synchronisation

- La notion de synchronisation précise la **nature de la communication**, et les **règles** qui régissent le passage des messages
- Message simple**
 - L'expéditeur reste bloqué jusqu'à ce que le destinataire réponde au message
 - Cas classique d'appel d'une méthode
- Message synchrone**
 - L'expéditeur reste bloqué jusqu'à ce que le destinataire accepte le message (réponse par un autre message)

Franck Ledoux DESS Bio-Informatique 2003-2004 68

Relations entre objets : synchronisation

- Message asynchrone**
 - L'envoi du message ne bloque pas l'expéditeur, celui-ci ne sait même pas si le message sera traité...
- Message minuté (ou borné)**
 - Expéditeur bloqué jusqu'à acceptation du message par le destinataire à concurrence d'une durée donnée
- Message débordant**
 - Déclenchement d'une opération à la réception uniquement si le destinataire s'est mis en attente de message. L'expéditeur est libéré dès l'envoi

Franck Ledoux DESS Bio-Informatique 2003-2004 69

1^{er} bilan : approche OO et qualité logicielle

Apports

- Encapsulation**
 - Consiste à séparer les aspects externes d'un objet, accessibles par les autres objets, des détails de son implantation interne
 - Équilibre traitements / données
 - Modularité gérée harmonieusement
- Abstraction**
 - Consiste à se concentrer sur les aspects essentiels, inhérents d'une entité et à en ignorer les propriétés accidentelles
 - Adaptabilité
 - Réutilisabilité / portabilité
 - Traçabilité

Franck Ledoux DESS Bio-Informatique 2003-2004 70

Classe

- Une classe est une définition d'un type d'objets
- Elle décrit un groupe d'objets ayant les **mêmes propriétés** et le **même comportement** afin d'en faciliter la gestion
- Classe = **maquette** d'objets

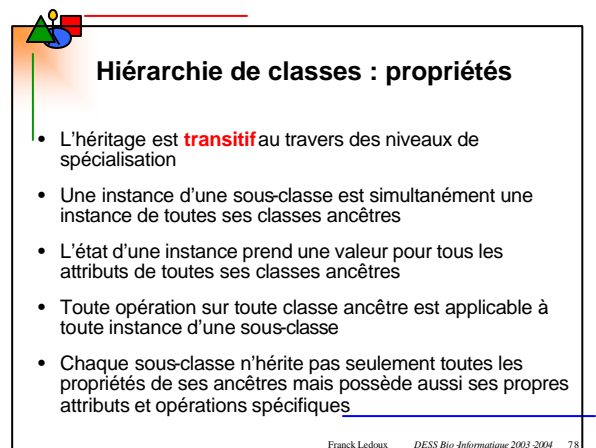
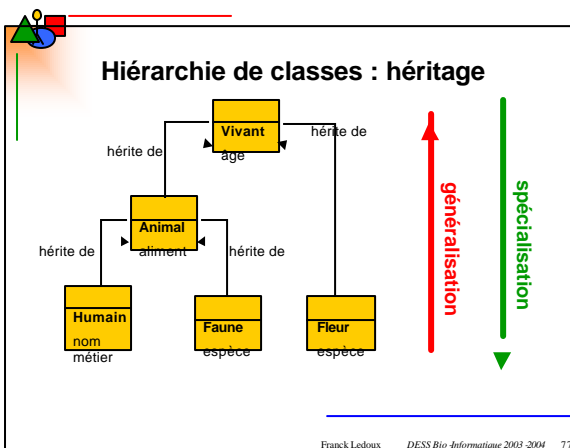
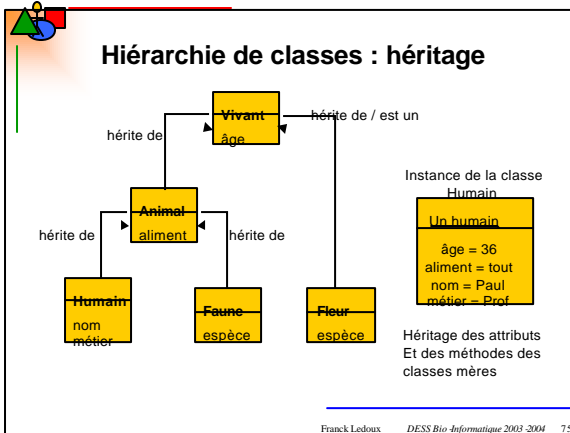
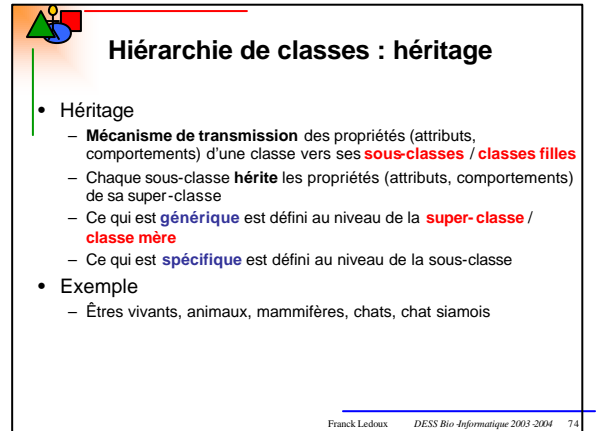
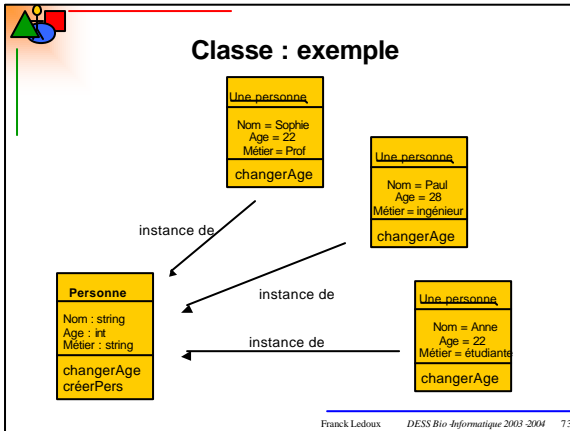
Abstraction du problème pour en réduire la complexité

Franck Ledoux DESS Bio-Informatique 2003-2004 71

Classe : instantiation

- Instance**
 - Tout objet d'une classe est appelé instance de la classe
 - La classe décrit la structure de ses instances : elles auront les mêmes attributs et méthodes que la classe
 - Mais chaque instance à ses propres valeurs d'attributs
- Cycle de vie d'une instance**
 - Création d'instance : constructeur – méthode de classe qui ne sera pas dans le comportement de l'instance
 - L'état courant d'une instance est défini en contexte et en toute indépendance par l'objet créé

Franck Ledoux DESS Bio-Informatique 2003-2004 72



Hérarchie de classes : propriétés

- La notion d'héritage est différente pour les attributs et les méthodes
 - Attributs : ils sont hérités tels quel sans modification
 - Méthodes : une méthode héritée peut être redéfinie dans la sous-classe
- Exemple :
 - Article Obtenir_Prix_TTC {renvoie Prix_HT*1.055}
 - Article_de_Luxe Obtenir_Prix_TTC {renvoie Prix_HT***1.196**}

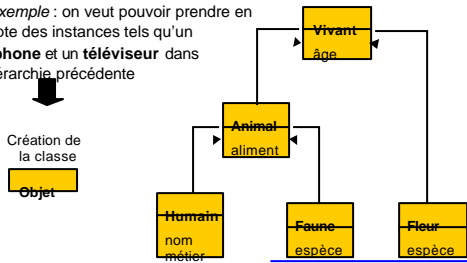
Classe abstraite

- Classe ne permettant pas d'instancier d'objets
- Une telle classe possèdent des **sous-classes concrètes**, i.e. instanciables
- Elle sert à factoriser des attributs et des comportements communs, même si ceux-ci ne sont totalement définis que dans les sous-classes concrètes.
- Exemple
 - Classe Vivant
 - Classe Animal
- Utilité
 - Représentation de concepts fondamentaux pour l'application



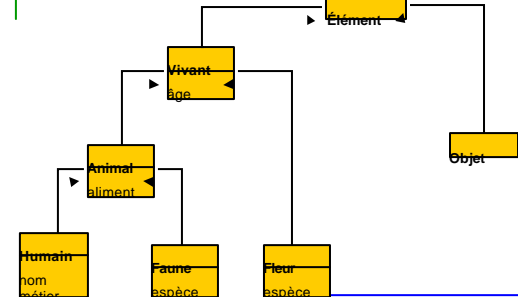
Hérarchie de classe : problématique

- Un même problème peut se modéliser de diverses façons
- Problématique de l'analyse / conception
- Exemple : on veut pouvoir prendre en compte des instances tels qu'un **téléphone** et un **téléviseur** dans la hiérarchie précédente



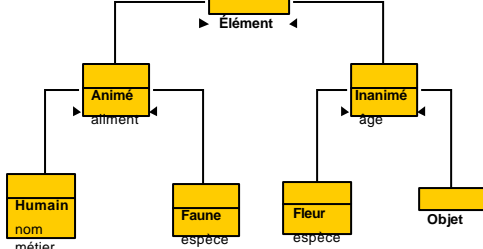
Hérarchie de classe : problématique

- Première solution



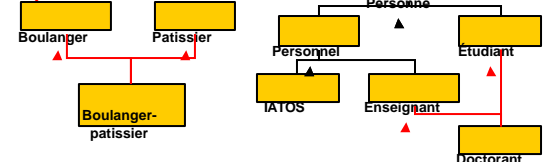
Hérarchie de classe : problématique

- Seconde solution : remise en cause de la hiérarchie



Héritage multiple

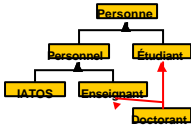
- Un héritage est **simple** lorsqu'une classe n'hérite que d'une **seule** autre classe
- Un héritage est **multiple** lorsqu'une classe hérite de **plusieurs** autres classes



- Les héritages multiples permettent souvent d'avoir **différents points de vue** sur les classes

Héritage multiple

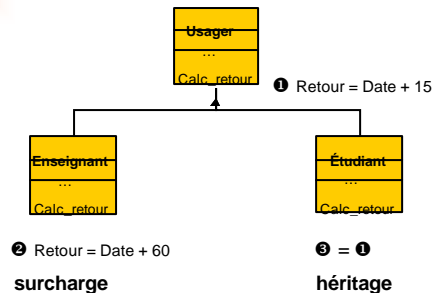
- Conflits entre propriétés héritées
 - Héritage d'un même nom d'attribut (ou de méthodes) par 2 sur-classes
 - Exemple : filières d'inscription et d'enseignement
 - Solution : interdiction de conflits de noms d'attributs, priorités entre classes
 - Si plusieurs « chemin » d'héritage existe entre CI-b et CI-a, CI-b hérite plusieurs fois des caractéristiques de CI-a
 - Exemple : Doctorant hérite 2 fois de Personne
 - Solution : supprimer des liens d'héritage, ou résoudre les problèmes de conflit de nom s'ils ont lieu



Polymorphisme

- **Surcharge**
 - Définition : redéfinition d'une méthode héritée pour pouvoir lui donner une implantation différente
 - Utilité : la méthode garde la même sémantique – spécialisation tout en gardant le même comportement vis à vis de l'extérieur : méthode polymorphe
- Polymorphisme : **liaison dynamique**
 - Elle se produit lors de l'envoi d'un message à un objet : l'objet réagit en recherchant la méthode à partir de sa classe puis en remontant dans la hiérarchie de ses super-classes
 - Conséquences : déclenchement de traitements différents suivant le contexte (i.e. classe réceptrice)

Polymorphisme : exemple BD



Polymorphisme

- Avantages
 - **Conception** : un seul nom de méthode pour des traitements équivalents mais spécifiques
 - **Adaptabilité** : nouveau besoin = nouvelle sous-classe avec surcharge pour adaptation à ce besoin spécifique

Bilan sur l'approche OO

- Structuration
 - Découpage des programmes en **entités informatiques** dont l'existence est directement liée aux **entités « réelles »**
 - **Arbre** de classification par héritage
- Modularité
 - Regroupement au sein de la même entité des attributs et méthodes qui sont logiquement liés, ce qui facilite la compréhension
- Réutilisabilité
 - Définition d'une nouvelle classe par **agrégation** d'autres classes
 - Définition d'une nouvelle classe par **héritage** d'autres classes
- Évolutivité
 - **Indépendance des clients** vis-à-vis de l'implantation grâce à l'encapsulation
 - Grâce au polymorphisme de méthodes et à la liaison dynamique, **préservation du code factorisé** pour de nouvelles classes d'objets

Introduction à UML Unified Modeling Language

- Franck Ledoux
- DESS Bio-Informatique 2003 -2004*
- 91

- 

Franck Ledoux *DESS Bio-Informatique 2003-2004* 92

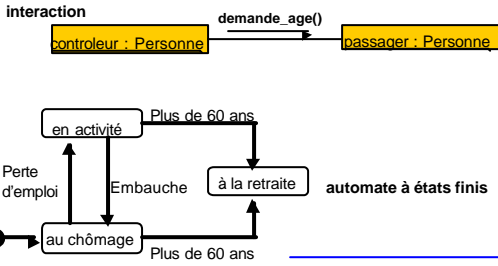
- Franck Ledoux
- DESS Bio-Informatique 2003-2004*
- 93

Franck Ledoux *DESS Bio-Informatique 2003 -2004* 95

- Franck Ledoux
- DESS Bio-Informatique 2003-2004*
- 96

Terminologie UML

- **Comportementaux** – parties dynamique (interactions, automates à états finis)



Terminologie UML

- **De regroupement** – parties organisationnelles, ce sont les boîtes qui compose le modèle (paquetages, frameworks, sous-systèmes)

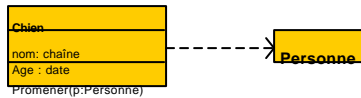


- **D'annotation** – parties explicatives, ce sont les commentaires qui peuvent accompagner tout élément à des fins d'explication, de description ou de remarque (notes)



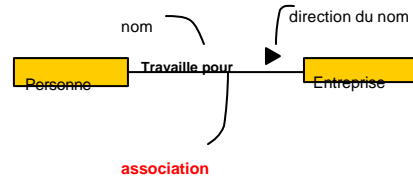
Terminologie UML

- Les **relations** (liens entre éléments) sont :
 - **La dépendance** – relation sémantique entre 2 éléments selon laquelle un changement apporté à l'un (elt indépendant) peut affecter la sémantique de l'autre (elt dépendant)



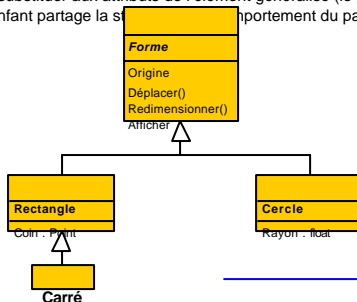
Terminologie UML

- **L'association** – relation structurelle qui décrit un ensemble de liens, un lien constituant une relation entre différents objets



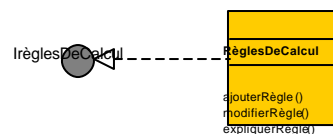
Terminologie UML

- **La généralisation** – relation de spécialisation / généralisation selon laquelle les attributs de l'élément spécialisé (l'enfant) peuvent se substituer aux attributs de l'élément généralisé (le parent). L'enfant partage la sémantique du parent



Terminologie UML

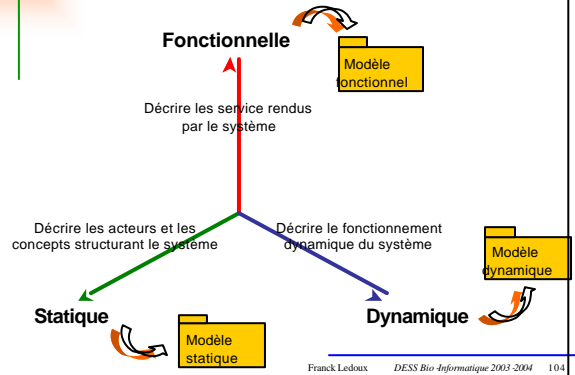
- **La réalisation** – relation sémantique entre classificateurs, selon laquelle un classificateur spécifie un contrat dont l'exécution est garantie par un autre classificateur



Terminologie UML

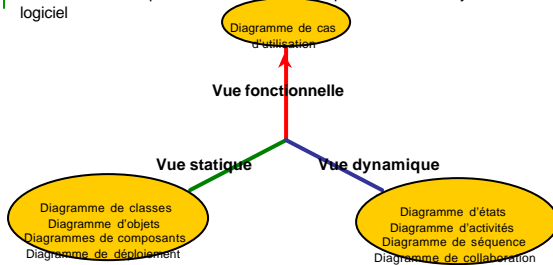
- Les **diagrammes** (représentation graphique d'un ensemble d'éléments qui constituent un système)
 - Servent à visualiser un système sous **différentes perspectives**
 - Pour un système complexe, ils peuvent ne représenter qu'une **vue partielle** du système
 - Sont classés selon **trois vues principales** du système

3 vues de modélisation



9 diagrammes en UML

- UML s'articule autour de plusieurs types de diagrammes, chacun d'eux étant dédié à la représentation de concepts particuliers d'un système logiciel

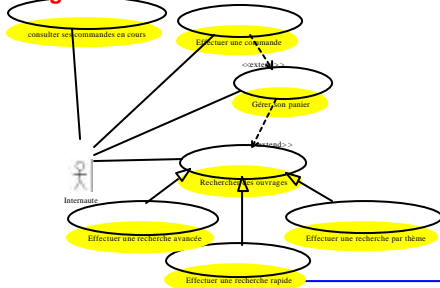


Vue fonctionnelle

- Diagramme de cas d'utilisation**
 - inventé par Ivar Jacobson
 - utilisé essentiellement dans l'activité de spécification des besoins
 - décrit selon le point de vue de ses futurs utilisateurs le comportement du système
 - décrit des relations entre le système et son environnement
 - présente la vue statique des cas d'utilisation d'un système

Vue fonctionnelle

- Diagramme de cas d'utilisation**

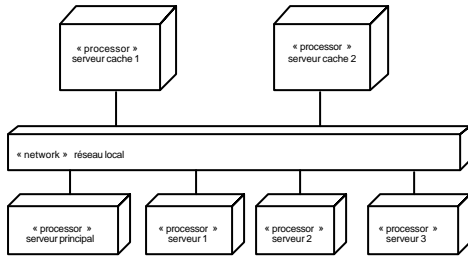


vue statique ou structurale (1/4)

- Diagramme de classes**
 - point central dans un développement objet
 - exprime la structure statique du système
 - en **analyse**, il décrit la structure des entités manipulées par les utilisateurs
 - en **conception**, il représente la structure du code orienté objet

Vue statique ou structurelle (4/4)

• Diagramme de déploiement



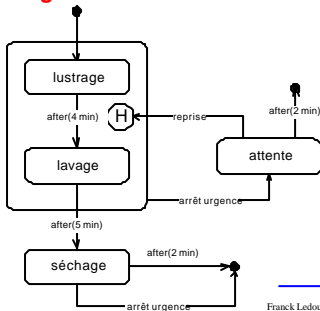
Vue dynamique ou comportementale (1/4)

• Diagramme d'états-transitions

- représente le **cycle de vie** commun aux objets d'une même classe
- **automates à états finis** composés d'états, de transitions entre états, d'événements et d'activités
- Important pour la représentation du comportement d'une interface, d'une classe ou d'une collaboration
- met l'accent sur le comportement d'un objet **ordonné par les événements**
- complète la connaissance des classes en analyse et en conception

Vue dynamique ou comportementale (1/4)

• Diagramme d'états-transitions



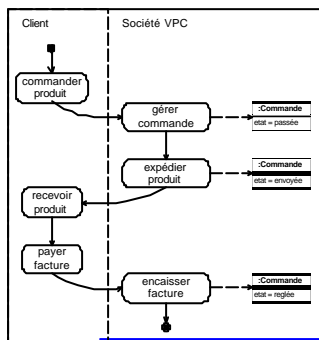
Vue dynamique ou comportementale (2/4)

• Diagramme d'activités

- Type particulier de diagramme d'états-transitions
- Décrit la succession des activités au sein d'un système
- Met l'accent sur le flot de contrôle entre objets.
- représente les règles d'enchaînement des activités dans le système
- fournit la structure d'une opération en action

Vue dynamique ou comportementale (2/4)

• Diagramme d'activités



Vue dynamique ou comportementale (3/4)

• Diagramme de séquence

- **diagramme d'interaction**
- représente les **échanges de messages entre objets**, dans le cadre d'un fonctionnement particulier du système
- Mettent l'accent sur le **classement chronologique** des messages
- sert à développer les scénarii d'utilisation du système en **analyse** : il représente les interactions temporelles entre objets dans la réalisation d'une **interface Homme-Système**
- en **conception**, il représente les interactions temporelles entre objets dans la réalisation d'une **opération**

Vue dynamique ou comportementale (3/4)

- Diagramme de séquence

```

sequenceDiagram
    participant client
    participant transaction as transaction
    participant pODBQuery as pODBQuery

    client->>transaction: create (transient)
    activate transaction
    client->>transaction: setActions(a,d,o)
    deactivate transaction
    transaction->>pODBQuery: setValues(d,3,4)
    activate pODBQuery
    deactivate pODBQuery
    transaction->>pODBQuery: setValues(a,"CO")
    activate pODBQuery
    deactivate pODBQuery
    transaction->>client: ...collected...
    deactivate transaction
    client->>transaction: destroy
    destroy transaction
  
```

Franck Ledoux DESS Bio-Informatique 2003-2004 121

Vue dynamique ou comportementale (4/4)

- Diagramme de collaboration

- diagramme d'interaction
- représente les échanges de messages entre objets, dans le cadre d'un fonctionnement particulier du système
- met l'accent sur l'**organisation structurelle** des objets qui envoient et reçoivent des messages
- permet de concevoir et de placer les méthodes sur les classes appropriées
- par rapport au diagramme de séquence, on dispose distingue la structure statique du modèle, mais on voit moins bien les échanges de messages

Franck Ledoux DESS Bio-Informatique 2003-2004 122

Vue dynamique ou comportementale (4/4)

- Diagramme de collaboration

```

classDiagram
    class client
    class transaction as transaction
    class pODBQuery as pODBQuery

    client --> transaction : 1: « create »
    client --> transaction : 2: setActions(a,d,o)
    client --> transaction : 3: « destroy »
    transaction --> pODBQuery : 2.1: setValues(d,3,4)
    transaction --> pODBQuery : 2.2: setValues(a,"CO")
  
```

Franck Ledoux DESS Bio-Informatique 2003-2004 123

conclusion

Franck Ledoux DESS Bio-Informatique 2003-2004 124

Présentation du UP Unified Process

Franck Ledoux DESS Bio-Informatique 2003-2004 125

Introduction

- Processus
 - Ensemble de directives ayant pour objectif la réalisation ou l'évolution d'un logiciel
 - Chaque directive définit « qui » fait « quoi » et « à quel moment »
- Unifié
 - Issu de différentes approches de processus
 - Objectory (Jacobson) : les cas d'utilisation
 - Approche Rational :
 - Approche par composants
 - Approche itérative
 - Issu de la notation UML

Franck Ledoux DESS Bio-Informatique 2003-2004 126

Introduction

- UP (Unified Process) est une méthode générique de développement de logiciel
 - Générique** – il est nécessaire d'adapter UP au contexte du projet, de l'équipe, du domaine et/ou de l'organisation
 - RUP : Rational Unified Process (version 5.0 - 1998)
 - XUP : eXtreme Unified Process
- Cadre général
 - Regroupe les activités à mener pour transformer les besoins d'un utilisateur en système




Caractéristiques essentielles

- Orienté **composants**
- Utilise le langage de modélisation **UML**
- Piloté par les **risques**
- Piloté par les **cas d'utilisation**
- Centré sur l'**architecture**
- Itératif** et **incrémental**

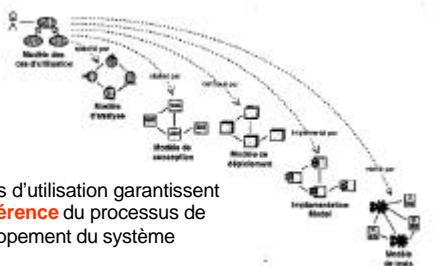
Piloté par risques

- Première cause de risque à éliminer : incapacité de l'architecture à répondre aux contraintes opérationnelles
- Les risques majeurs du projet doivent être identifiés au plus tôt
- Et levés le plus rapidement possible

Piloté par les cas d'utilisation

- Objectif principal d'un système : **rendre service à ses utilisateurs** →  comprendre leurs désirs et besoins
- Utilisateur = humain ou autre système logiciel
- Les cas d'utilisation expriment les **exigences fonctionnelles** des utilisateurs
- Les cas d'utilisation sont utilisés tout au long du processus :
 - pour la spécification des besoins du système
 - pour guider le processus à travers l'utilisation des diagrammes UML

Piloté par les cas d'utilisation



Les cas d'utilisation garantissent la **cohérence** du processus de développement du système

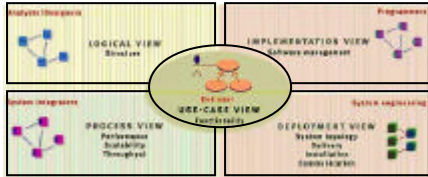
→ Développés en tandem avec l'architecture du système

Centré sur l'architecture

- Vue sur l'architecture dès le démarrage du PU
- L'architecture subit l'influence
 - des besoins, vœux de l'utilisateur
 - de la plate-forme sur laquelle devra s'exécuter le système
 - des composants réutilisables disponibles
 - des considérations de déploiement, les systèmes existants
 - des besoins non fonctionnels (performance, fiabilité,...)
- L'architecture doit être
 - Évolutive, modulaire
 - Basée autour d'une architecture de référence

Centré sur l'architecture

- La conception de l'architecture est
 - Itérative
 - Cadrée par les cas d'utilisation principaux. Elle doit donc prévoir la réalisation de tous les cas d'utilisation
- Décrite comme les différentes vues du système à construire (modèle des 4+1)



Modèle des 4+1

- Ces cinq vues sont indépendantes les unes des autres, ce qui permet aux différents intervenants de se concentrer sur les problèmes de l'architecture du système qui les concernent le plus.
- Elles interagissent également entre elles— les nœuds de la vue de déploiement comprennent des composants de la vue d'implémentation, qui, à leur tour, correspondent à la réalisation physique de classes, d'interfaces, de collaborations et de classes actives dans les vues de conception et de processus.
- UML permet de représenter chacune de ces cinq vues ainsi que leurs interactions.

Processus itératif et incrémental

- Le Processus Unifié prône la **division** d'un grand projet en une succession de petits projets
- Cette division est **transparente** pour le client
- Chaque **itération** constitue un **mini-projet** de courte durée (environ 1 mois)
- À la fin de chaque itération, une partie exécutable du système final est produite, de façon incrémentale (par ajout)
- Croissance du système par itérations successives, feedback et adaptations cycliques

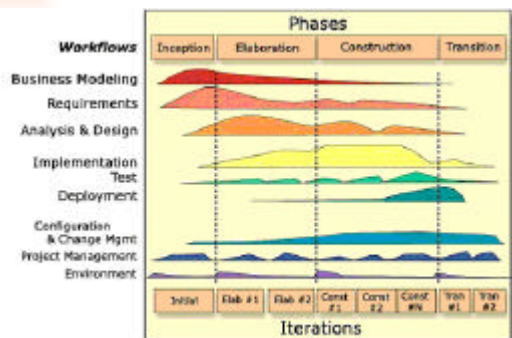
Processus itératif et incrémental

- Avantages d'un processus itératif contrôlé**
 - Limiter les **coûts** aux strictes dépenses liées à une itération
 - Limiter les **risques de retard** par une identification des problèmes dès les premiers stades du développement
 - Accélérer le rythme de développement grâce à des **objectifs clairs et à court terme**
 - Prise en compte des besoins des utilisateurs au cours du développement (pas intégralement défini initialement)
- L'architecture fournit la structure qui servira de cadre au travail effectué au cours des itérations, tandis que les cas d'utilisation définissent les objectifs et orientent le travail de chaque itération

Cycle de vie du Processus Unifié

- Répétition un certain nombre de fois d'une série de cycles
- Tout cycle se conclut par la livraison d'une version du produit au client
- Un cycle s'articule en 4 phases :
 - L'inception ou création (étude d'opportunité)
 - L'élaboration (architecture, planification)
 - La construction
 - La transition
- Ses activités de développement sont définies par 5 **workflows** fondamentaux qui décrivent :
 - La capture des exigences
 - L'analyse et la conception
 - L'implantation
 - Le test
 - Le déploiement

Phases et itérations





Inception ou création

- Objectifs :
 - Définir la « vision » du projet, sa portée, sa faisabilité, son « business case » pour décider au mieux de sa poursuite ou de son arrêt
 - Enveloppe globale des coûts, étude de rentabilité,...
 - Cas d'utilisation principaux
 - Première architecture de référence compatible avec les cas d'utilisation principaux
- Jalons :
 - Les parties s'accordent sur les objectifs
 - Les exigences décrites par les cas d'utilisation principaux sont bien comprises
 - Les estimations des coûts, des délais, des risques sont raisonnables
 - Le prototype couvre correctement les exigences



Élaboration

- 3 objectifs principaux :
 - Identifier et décrire la majeure partie des besoins des utilisateurs
 - Construire (et pas seulement décrire dans un document !) l'architecture de base du système
 - Lever les risques majeurs du projet
- Jalons :
 - Le cahier des charges est-il valide ?
 - L'architecture est-elle stable ?
 - Le plan de la phase de construction est-il suffisamment détaillé ?
 - Les estimations qu'il contient sont-elles vraisemblables ?



Construction

- Phase la plus consommatrice en ressources et en efforts
- Objectifs :
 - Concevoir et implanter l'ensemble des éléments opérationnels (autres que ceux de l'architecture de base)
 - Obtenir un logiciel de qualité satisfaisante aussi rapidement que possible
 - Mettre au point les versions alpha, bêta et autres versions de test
- Jalons :
 - La version bêta du logiciel livrée chez les utilisateurs est-elle au point et répond-elle aux exigences ?
 - Les parties concernées sont-elles prêtes pour la mise en place du logiciel chez les utilisateurs ?
 - Les dépenses réelles sont-elles acceptables



Transition

- Objectifs :
 - Livrer une version finale du logiciel aux utilisateurs finaux (conversion de données, déploiement,...)
 - Rendre les utilisateurs opérationnels et autonomes (formation,...)
- Jalons :
 - Les utilisateurs sont-ils satisfaits ?
 - Les dépenses réelles sont-elles acceptables par rapport à celles prévues ?



Processus unifié simplifié (PUS)

- Simplification du PU
 - Retrait des itérations
 - Utilisation d'une partie des diagrammes UML
- Mis en œuvre par P. Roques dans « *UML, modéliser un site e-commerce* »
- Approche suivie dans ce cours



Caractéristiques du PUS

- Conduit par les **cas d'utilisation**, comme UP, mais beaucoup plus simple
- Relativement **léger et restreint**, comme XP, mais sans négliger les activités de modélisation en analyse et conception
- Fondé sur un sous-ensemble nécessaire et suffisant du langage UML :
 - Diagramme de cas d'utilisation
 - Diagrammes d'interaction (séquence et collaboration)
 - Diagramme de classe
 - Diagramme d'activité
 - Diagramme d'états-transitions

Diagramme de cas d'utilisation

Diagramme de cas d'utilisation

- Inventé par Ivar Jacobson (use cases)
- Description du point de vue de l'**extérieur** du **comportement** du système en utilisant des actions/réactions



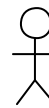
- Décrit **ce que fait** le système **et non comment** il le fait
- Définitions des **limites** et des **objectifs** du système
- Description des relations entre le système et son environnement
- Utilisation essentielle pour **spécifier les besoins**

Comportement du système

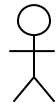
- Quelles fonctionnalités doivent être fournies par le système ?
- Les cas d'utilisation :
 - Les fonctions du système (cas d'utilisation)
 - Les limites (les acteurs)
 - Les relations entre les cas et les acteurs (diagramme de cas d'utilisation)
- Les cas d'utilisation servent à communiquer autant pour les usagers que pour les développeurs

Acteurs

- Acteur : entité **externe** qui **agit** sur le système
 - prend les décisions contrairement à un élément logiciel
 - possède un rôle par rapport au système
 - fournit de l'information en entrée
 - et/ou reçoit de l'information en sortie
 - soit un utilisateur
 - soit un autre système



Rôle de l'acteur



Étudiant

Comment identifier les acteurs ?

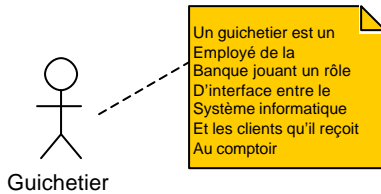
- Qui est intéressé par un certain besoin ?
- Où le système est-il utilisé dans l'organisation ?
- Qui bénéficiera de l'utilisation du système ?
- Qui fournira au système l'information, qui l'utilisera et la maintiendra ?
- Qui va supporter et maintenir le système ?

Acteurs vs utilisateurs

- Ne pas confondre acteur et personne utilisant le système :
 - Une même personne peut jouer plusieurs rôles
 - Plusieurs personnes peuvent jouer un même rôle
 - Un acteur n'est pas forcément une personne physique
- **Types d'acteurs** :
 - Utilisateurs **principaux** : personnes qui utilisent les fonctions principales du système
 - Utilisateurs **secondaires** : personnes qui effectuent des tâches administratives ou de maintenance
 - Matériel **externe** : dispositifs matériels faisant partie du domaine de l'application
 - Autre systèmes

Définition des acteurs

- Pour chaque acteur :
 - Choisir un identificateur représentatif du rôle
 - Éventuellement accompagnée d'une brève description textuelle



Cas d'utilisation

- Un cas d'utilisation :
 - correspond à une **manière spécifique d'utiliser le système**
 - modélise un **dialogue** entre un utilisateur et le système
- Représentation d'une **fonctionnalité** offerte par le système
- L'ensemble des cas d'utilisation forme toutes les façons que le système pourra être utilisé

Comment identifier les CU ?

- Quelles sont les tâches de chaque acteur ?
- Est-ce qu'un acteur crée, enregistre, modifie, enlève ou consulte une information dans le système ?
- Quel cas d'utilisation sera responsable de ces tâches ?
- Quel cas d'utilisation supportent ou maintiennent le système ?

Cas d'utilisation : représentation

- En UML, un cas d'utilisation est représenté par un ovale

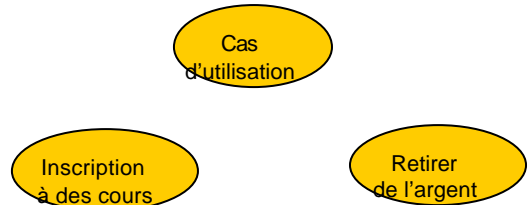
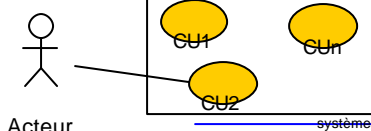


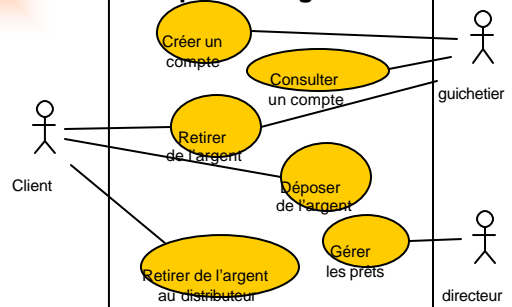
Diagramme de cas d'utilisation

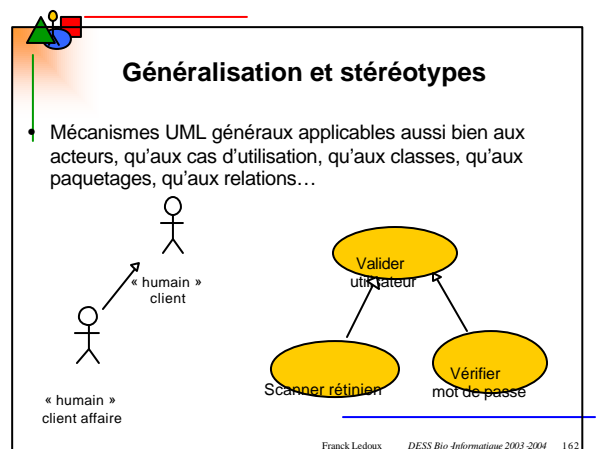
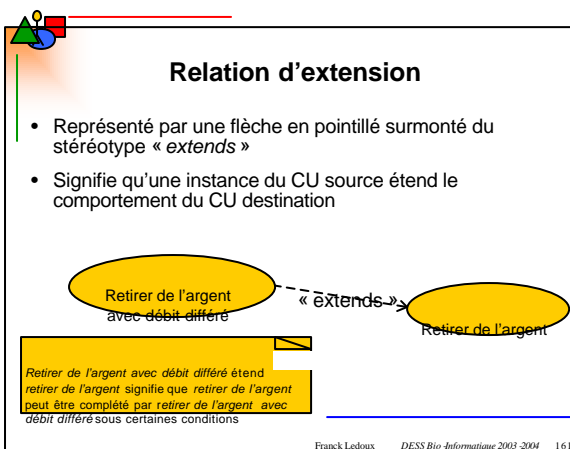
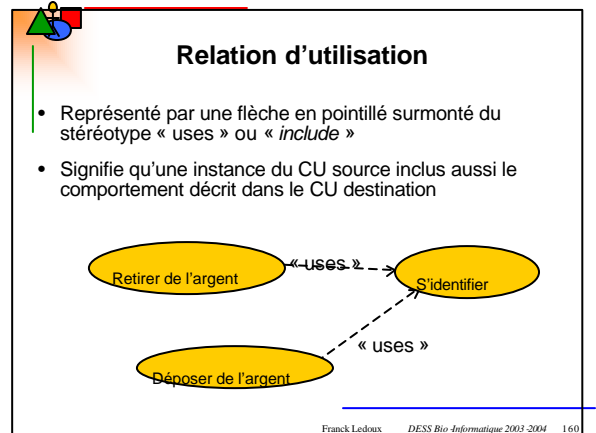
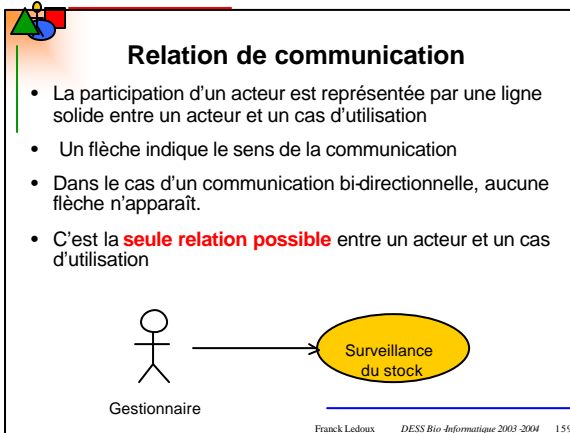
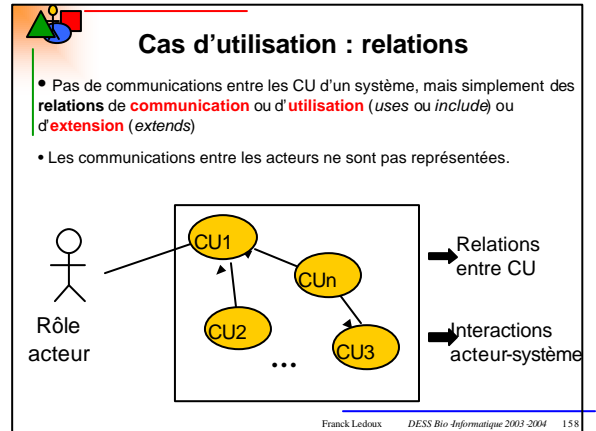
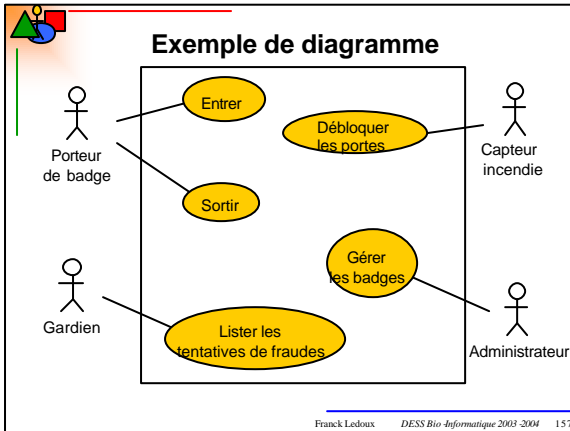
Cas d'utilisation : ensemble des actions réalisées par le système en réponse à une action d'un acteur

- Suite d'interactions entre un acteur et le système
- Correspond à une fonction visible par l'utilisateur
- Permet d'atteindre un objectif aux yeux de l'utilisateur
- Doit être utile
- Les cas d'utilisations ne doivent pas se chevaucher



Exemple de diagramme

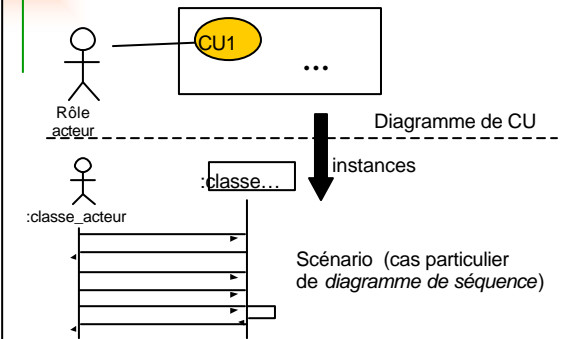




Cas d'utilisation et scénario

- Le **système** = **ensemble** de **cas d'utilisation**
- Le système possède les cas d'utilisation mais pas les acteurs
- Un cas d'utilisation = ensemble de « chemins d'exécutions » possibles
- Un **scénario** = un chemin particulier d'exécution
= séquence d'événements
= **instance** de cas d'utilisation

Cas d'utilisation vs scénario



Cas d'utilisation et scénario

- Spécification **exhaustive** de tous les scénarii difficile, voire impossible
- Sélection des scénarii les plus intéressants
 - Scénario **optimal** : décrit l'interaction la plus fréquente
 - Scénarii **dérivés** : décrit certaines alternatives importantes non décrites dans le scénario optimal

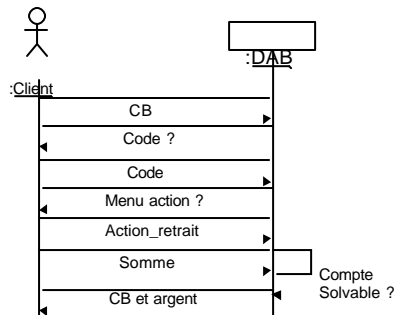
Cas d'utilisation et scénario

Un **scénario** peut être représenté par un diagramme de **séquence** qui décrit un échange particulier entre un ou plusieurs acteurs et le système

- Nature des infos **échangées** entre les **instances** d'acteurs ou d'objets du système
- Aspect **temporel** : flot **ordonné d'événements**

Un scénario peut également être représenté par un diagramme de collaboration

Cas d'utilisation vs scénario



Attention !

- Les cas d'utilisation ne sont qu'une vue externe en non un contrôleur procédural
- Le logiciel est souvent structuré d'une manière totalement différente des cas d'utilisation
- Les cas d'utilisations ne sont qu'une certaine vision du logiciel !
- Ils servent à spécifier les besoins

Construction d'un diagramme de CU

- Recenser les acteurs et les cas d'utilisation
- Structurer en paquetage
- Étudier les relations entre cas d'utilisation
- Classement des cas d'utilisation par niveau de priorité
- Décrire ensuite un CU en écrivant des scénarii sous forme textuelle ou avec un diagramme de séquence
- Faire un sommaire des scénarii principaux et des CU
- S'assurer de la cohérence finale

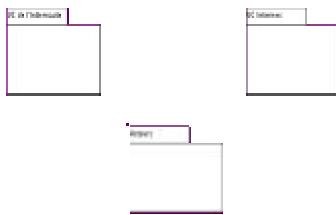
Recensement des C.U.

- Ne pas être trop « atomique »
- Un C.U. = un ensemble de séquences d'utilisation connexes



Structuration en paquetages

- Séparer en ensembles fonctionnels cohérents
 - Paquetages (packages)
 - Relations de dépendances (idéal : un seul sens)



Étude des relations entre CU



Classement des C.U.

- Prise en compte
 - des priorités client (ne pas tout classer en « haut »)
 - des risques techniques
- Itérations

Cas d'utilisation	Priorité	Risque	Itération
Rechercher des ouvrages	Haute	Moyen	2
Gérer son panier	Haute	Bas	3
Effectuer une commande	Moyenne	Haut	4
Consulter ses commandes en cours	Basse	Moyen	6
Consulter l'aide en ligne	Basse	Bas	7
Maintenir le catalogue	Haute	Haut	1
Maintenir le site	Moyenne	Bas	5

Diagramme de séquence

Diagramme de séquence

- Notation dérivée des « Object Message Sequence Charts » du Siemens Pattern Group
- Description des interactions entre les objets composant le système
- Représentation se concentrant sur le point de vue temporel
- Adapté à la modélisation des aspects dynamiques des systèmes temps réel et des scénarii complexes mettant en œuvre peu d'objets
- C'est un diagramme d'interaction au même titre que les diagrammes de collaboration

Interactions dans le système

- Une interaction se traduit par l'envoi d'un message entre objets
- Les diagrammes de séquences comportent :
 - les objets intervenant dans l'interaction (acteurs ou objets appartenant au système)
 - la description de l'interaction (messages)
 - les interactions entre les intervenants (diagramme de séquences)
- Les diagrammes de séquence servent à communiquer autant pour les usagers que pour les développeurs

Utilisation

- Documentation des cas d'utilisation :
 - Description des interactions en des termes proches de l'utilisateur
 - Étiquettes des messages correspondent à des événements se produisant dans le système
- Représentation des interactions « informatiques » et répartition des flots de contrôle :
 - Le concept de message unifie les formes de communication entre objets (appel de procédure, événement discret, signal,...)

Éléments d'un diag. de séquence

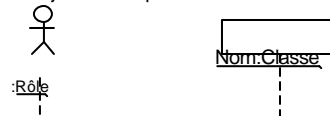
- Objets
- Lignes de vie
- Messages
- Contraintes temporelles
- Structures de contrôle
 - Boucles
 - Conditionnelles

Objets

- Les objets sont des entités appartenant au système ou situés à ses limites (acteurs)
- Ils représentent soit
 - des concepts abstraits ou
 - des acteurs (documentation des CU)
 - Des objets d'implantation (pour les interactions « informatiques »)
- Chaque objet représenté dans un diagramme de séquence correspond à une instance de classe
- On identifie les objets à partir des CU ou des diagrammes de classe

Objets : représentation

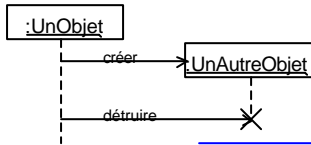
- En UML, les objets sont représentés comme suit :



- Le nom de l'objet est composé de son rôle (rôle ou nom) et/ou du nom de la classe instanciée (classe)
- Le nom doit être souligné pour indiquer qu'il s'agit d'une instance

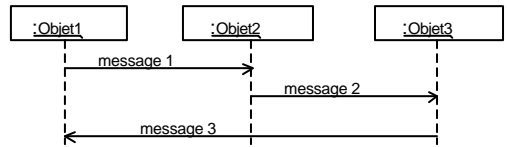
Ligne de vie des objets

- La ligne de vie est représentée par la ligne verticale en dessous des objets
- Elle représente la période durant laquelle l'objet existe
- Création d'un objet** : un message pointe sur le symbole de l'objet
- Destruction d'un objet** : sa ligne de vie se termine par une croix en trait épais



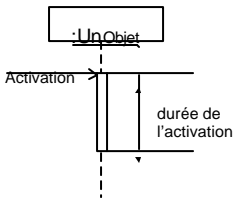
Messages

- Les objets communiquent en échangeant des messages représentés par des flèches
- Le nom du message ou du signal est porté par sa flèche
- L'ordonnancement horizontal des messages n'a pas de signification
- L'écoulement du temps se fait verticalement



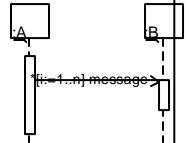
Activation des objets

- Une période d'activité correspond au temps pendant lequel un objet effectue une action directe ou indirecte
- Représentée par une bande verticale le long de la ligne de vie de l'objet



Étiquettes des messages

- Les étiquettes décrivent les messages auxquels elles se rapportent
- Syntaxe classique :
Garde itération résultat := nom_message arguments
- nom_message** : nom de l'opération ou du signal invoqué par l'intermédiaire de ce signal
- garde** : condition booléenne et optionnelle (représentée entre crochets) autorisant ou non l'envoi d'un message
- Itération**
 - Séquentielle : envoi séquentiel de n messages
*[clause d'itération]
 - Parallèle : envoi parallèle de n messages
*||[clause d'itération]

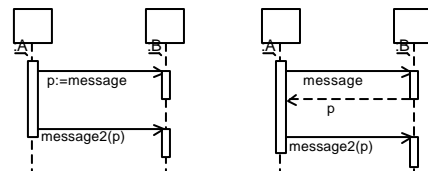


Étiquettes des messages

- Arguments**
 - Liste des paramètres du message séparés par des virgules
 - Les arguments et le nom de l'action déterminent sans ambiguïté l'action à réaliser
 - Les arguments peuvent contenir des valeurs retournées par des messages envoyés précédemment
- Exemples**
 - Afficher (x, y) : affiche les valeurs de x et y
 - Soustraire (aujourd'hui, dateDeNaissance) : calcule le nombre de jours entre deux dates

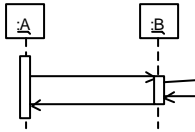
Étiquettes des messages

- résultat**
 - Le résultat est constitué d'une liste de valeurs retournées par le message
 - Ces valeurs peuvent être utilisées comme paramètres des autres messages



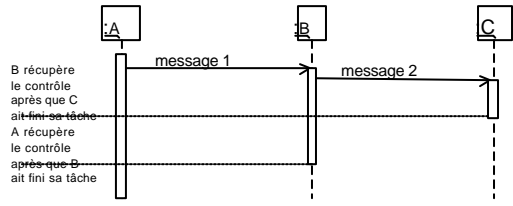
Flot de contrôle à plat

- Catégorie de messages utilisée pour indiquer la progression vers l'étape suivante d'une séquence
- Tous les messages de cette catégorie sont asynchrones
- Message représenté par une flèche
- Cas particulier : dans le cas de systèmes concurrents, une demi-flèche indique l'envoi d'un message et une flèche un message avec attente de prise en compte



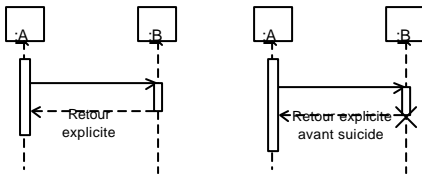
Appel de procédure

- Dans un appel de procédure (flot de contrôle emboîté), la séquence emboîtée doit se terminer pour que la séquence englobante reprenne le contrôle
- Les appels de procédures sont représentés par des flèches à pointe triangulaire



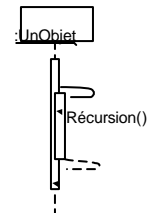
Retour explicite

- Dans le cas d'un système concurrent, il est utile d'expliciter la fin de l'exécution de sous-procédures
- On utilise une flèche pointillée (déjà utilisée dans le cadre de valeurs retournées)



Appel récursif

- Le cas particulier des envois de messages récursifs se représente par un dédoublement de la bande d'activation
- L'objet apparaît alors comme s'il était actif plusieurs fois



Contraintes temporelles

- Pour modéliser les délais de transmission non négligeables, on utilise :
 - une flèche oblique
 - ou des notations temporelles dans la marge
- Les instants d'émission et de réception d'un message sont représentés par le couple (symbole, symbole')

