M4207C
Application informatique dédiée aux R&T

Manuel Munier

Département RT
IUT des Pays de l'Adour
manuel.munier@univ-pau.fr

Plan du cours

Plan du cours

Ce que dit le PPN

- Objectifs du module
  - → Résoudre un problème depuis un cahier des charges succinct jusqu'à la mise à disposition d'une solution fonctionnelle
  - → Développer l'autonomie technique
- Compétences visées
  - → Autonomie des étudiants
  - → Renforcement des acquis
  - → Aisance de développement informatique et/ou télécoms et/ou réseaux
- Pré-requis
  - → Suivant la thématique de l'application, les modules réseaux/télécommunications ou informatique correspondants (de S1, S2 et S3)

Ce que dit le PPN

- Contenus
  - Développement d'une application réseaux et/ou télécommunication et/ou informatique
- Volume horaire : 27h
  - → Cours 6h / TD 6h / TP 15h
    (y compris exam & éval TP !!! ☹)
- Modalités de mise en œuvre
  - Travaux pratiques encadrés
  - Démonstration finale
- Mots clés
  - projet (transverse), autonomie

Déroulement

- Cours
  - Ce que nous allons faire :
    - ✓ introduction génie logiciel (cycle de vie,...)
    - ✓ introduction méthode de conception (qq schémas UML)
    - ✓ utilisation de l'environnement Eclipse
  - **Ce que nous ne ferons pas ! ! !**
    - ✗ Algorithmique : boucles, tests, tableaux, fonctions,...
    - ✗ Ligne de commande : gestion fichiers & répertoires,...
- TD/TP $\rightsquigarrow$ projet
  - fonctionnement en "mini projet"
  - équipes de 2-3 étudiants
  - analyse d'un cahier des charges
  - conception/programmation d'une appli
  - documentations (programmeur, utilisateur,...)
  - évolution de l'appli (ajout/modif d'une fonctionnalité)

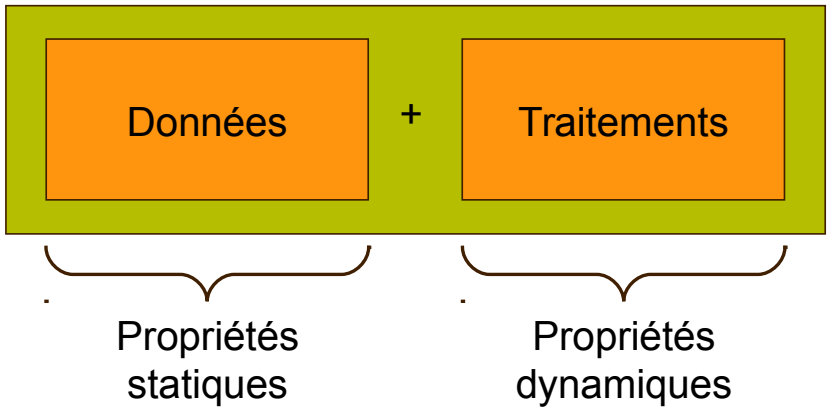
Quelques liens

- URL intéressantes
  - http://www.info.univ-tours.fr/~antoine/UML_GL.html
  - http://tisserant.org/cours/qualite-logiciel/
  - http://uml.free.fr/
  - http://uml.org/
  - http://www.umlet.com/

Plan du cours

Introduction

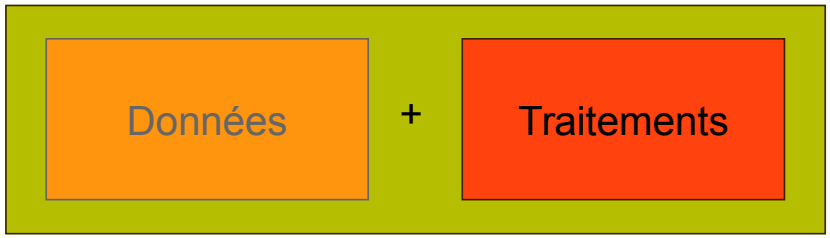
- Système informatique



  - données : variables, fichiers, bases de données,. . .
  - traitements : algorithmes, fonctions, méthodes, services,. . .

Introduction

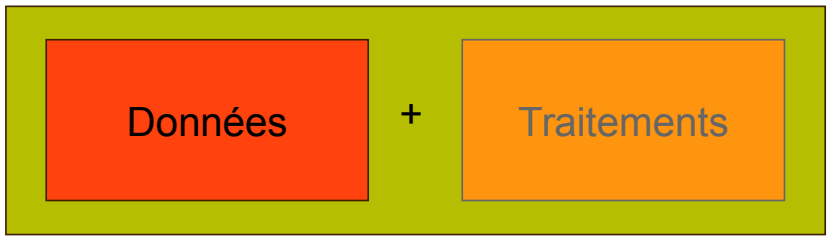
- Logiciels "classiques"



  - Les traitements constituent le plus souvent une part prépondérante dans la conception du logiciel

Introduction

- Systèmes d'information (bases de données)



  - Orientation données : "entrepôt de données"
  - Persistance des données
  - Systèmes actuels → part croissante aux traitements : "fouille de données" (data mining), aide à la décision (SIAD)
  - Arrivée du Big Data → focus sur les données

Vos compétences

M1207 Bases de la programmation

M2207 Consolidation des bases de la programmation

M2104 Bases de données

M2105 Web dynamique

\+ culture générale acquise en informatique, réseaux, système

Pourquoi un cours de 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 ingénierie : génie logiciel**

Objectifs de l'enseignement

- Introduction aux méthodes qui permettent de développer des systèmes logiciels complexes
- Vous allez détester... car vous n'êtes pas méthodiques
- Vous aller aimer... car proche de la "vie réelle" : développement en équipe et sur des projets importants

Génie logiciel

Une définition

Méthodologie de construction en équipe d'un logiciel complexe et à multiples versions.

- Programmation vs Génie Logiciel
  - programmation → activité personnelle
  - génie logiciel → activité en équipe
- Suivant les projets, la partie programmation (codage) ne représentera qu'entre 10% et 30% du coût total du projet

Logiciel : aspects économiques

- Importance économique du logiciel
  - importance croissante de l'informatique dans l'économie
  - coût du logiciel supérieur à celui du matériel
  - coût de la maintenance supérieur à celui de la conception

⇒ **Il faut améliorer la qualité du logiciel !**

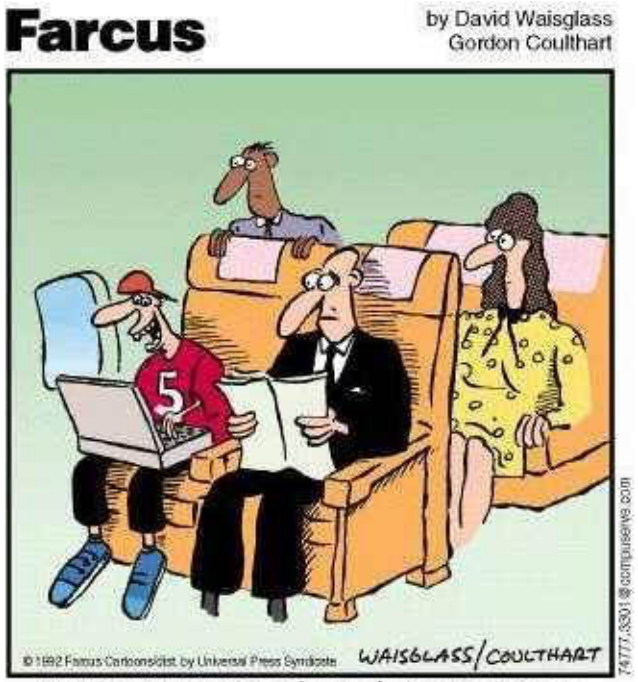
- Les pannes informatiques paraissent insupportables, les utilisateurs se sont habitués à une "informatique invisible" permanente.
- Des experts parlent d'un monde technologique que l'on ne peut plus débrancher. . .

La "crise" du logiciel

- Démarche ingénierie encore mal intégrée
- La non qualité des systèmes informatiques a des conséquences qui peuvent être très graves

NB : Les erreurs involontaires de conception et de codage représentent un tiers du coût des sinistres informatiques ! La malveillance quant à elle cause 60% de ce coût...

→ Licence Pro ASUR ☺



Yaahooo !!! J'ai réussi à faire planter le système de la tour de contrôle !!!

Pannes logicielles

- Convocation de centenaires à l'école
  - Convocation à l'école de personnes âgées de 106 ans
  - Cause : codage des années sur 2 caractères
- Mission Vénus
  - Passage à 5 000 000 de km de la planète, au lieu de 5 000 km prévus
  - Cause : remplacement d'une virgule par un point (au format US des nombres)
- 2 jours sans courant pour la station Mir, du 14 au 16 novembre 1997
  - Cause : plantage d'un ordinateur qui contrôlait l'orientation des panneaux solaires

Pannes logicielles

- Passage de la ligne
  - Au passage de l'équateur un F16 se retrouve sur le dos
  - Cause : changement de signe de la latitude mal pris en compte
- Y2K, le bug de l'an 2000
  - La lutte contre le bogue de l'an 2000 a coûté à la France 500 milliards de francs
  - Cause : la donnée "année" était codée sur 2 caractères, pour gagner un peu de place
- La carte bleue
  - Le secret des cartes bancaires reposait essentiellement sur un algorithme qui avait été publié sur un newsgroup !

Pannes logicielles

- Socrate
  - Système de réservation de places Socrate de la SNCF. Ses plantages fréquents, sa mauvaise ergonomie, le manque de formation préalable du personnel, ont amené un report important et durable de la clientèle vers d'autres moyens de transport.
  - Cause : rachat par la SNCF d'un système de réservation de places d'une compagnie aérienne, sans réadaptation totale au cahier des charges du transport ferroviaire.

Pannes logicielles

- Échec du premier lancement d'Ariane V
  - Au premier lancement de la fusée Ariane V, celle-ci a explosé en vol peu de temps après son décollage.
  - Cause : logiciel de plate forme inertielle repris tel quel d'Ariane IV sans nouvelle validation. Ariane V ayant des moteurs plus puissants s'incline plus rapidement que Ariane IV, pour récupérer l'accélération due à la rotation de la Terre. Les capteurs ont bien détecté cette inclinaison d'Ariane V, mais le logiciel l'a jugée non conforme au plan de tir (d'Ariane IV), et a provoqué l'ordre d'auto destruction. En fait tout se passait bien...
  - Coût du programme d'étude d'Ariane V : 38 milliards de Francs.

Pannes logicielles

- Perte de Mars Climate Orbiter, le 23 septembre 1999, après 9 mois de voyage
  - Coût : 120 M$
  - Cause : confusion entre pieds et mètres.

- Microsoft raciste ?
  - Le correcteur d'orthographe de Word proposait "anti-arabe" pour corriger "anti-stress"...

Pannes logicielles

- EBay indisponible
  - L'indisponibilité durant 22 heures du serveur web de EBay, site de vente aux enchères, a fait échouer plus de 2,3 millions d'enchères.
  - Dans un secteur où la compétitivité dépend plus que jamais du zéro-défaut face au consommateur, EBay a annoncé qu'elle remboursera les frais d'enregistrement des enchères en cours le jeudi 10 juin 1999, soit entre 3 et 5 millions de dollars. A Wall Street, le titre a chuté de plus de 9% (S&T Presse, 14 juin 1999).

Pannes logicielles

- Le micro-ordinateur menace-t-il la productivité ?
  - Des milliers d'heures de travail sont perdues à essayer de faire faire à l'ordinateur ce qu'il devrait faire, ou de comprendre des messages d'erreur incompréhensibles...

Pannes logicielles

- Ça plante !

Alors ?

- De façon générale, les programmes des élèves (et des profs) ne marchent jamais du premier coup !
- Pourtant ce sont des gens réputés intelligents ?
- Alors où est le problème ? Quelles sont les solutions ?

Arrêté ministériel du 22 décembre 1981

Un logiciel est défini comme "un ensemble de programmes, de procédés, de règles, de documentation relatifs au fonctionnement d'un ensemble de traitement de l'information".

Sûrs de vous ?

- Êtes-vous prêts à garantir la qualité des logiciels que vous écrivez ? Leur validité et leur fiabilité ? Pourriez-vous démontrer la qualité ?
- Pourquoi hésitez-vous ?

Rich Cook

"La programmation est aujourd'hui une course entre les ingénieurs informaticiens qui essaient de construire des programmes plus grands et mieux à l'épreuve des idiots, et l'univers qui essaie de produire des idiots plus grands et plus idiots. Jusqu'à présent, l'univers gagne."

Maîtriser le développement !

- Un nombre important de projets informatiques n'aboutissent pas aux logiciels répondant aux **besoins des utilisateurs** en respectant les **contraintes de budget** et de **délai**...

- *"90% des projets informatiques sortent en retard"* (Aberdeen)

Dérapages

- De nombreux projets informatiques n'ont jamais abouti
  - Système de réservation de places de United Airlines
  - Système d'exploitation Rhapsody
  - Avis Europe abandonne le déploiement de l'ERP Peoplesoft mené par Atos : 45M€ (LMI octobre 2004)

- *"30% des projets informatiques sont annulés avant la mise en production"* (Aberdeen)

Dérapages

- Ou n'ont pas réalisé les fonctionnalités attendues
  - Windows 3
  - Windows ME grand public
  - Windows Vista

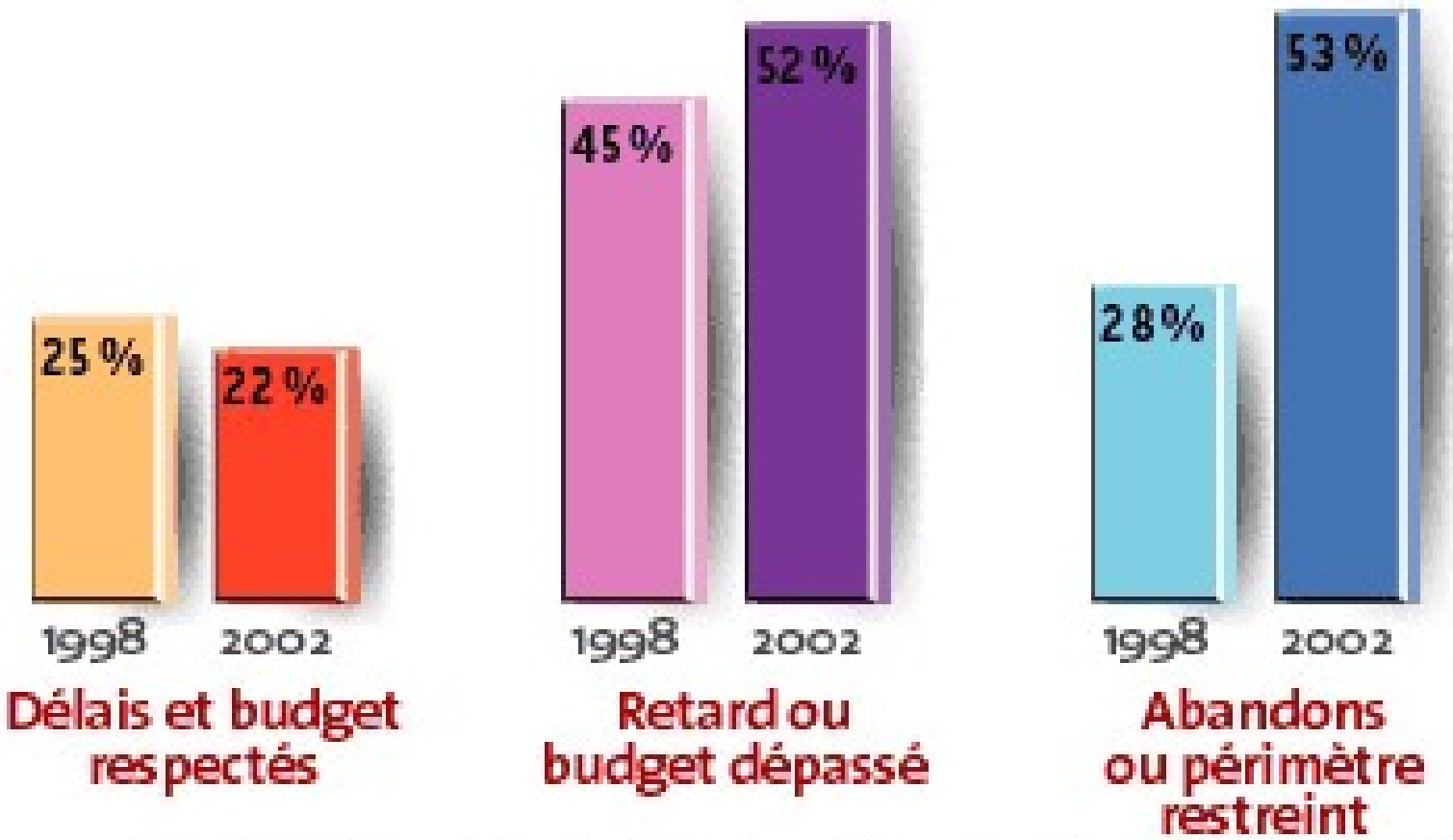
- *"50% des projets informatiques ne répondent pas au cahier des charges business"* (Gartner)

Dérapages

- Ou ont été des catastrophes économiques
  - Socrate
  - Informatisation des caisses de sécurité sociale, carte Vitale
  - Milieu 1999, le système d'information de la Bibliothèque Nationale de France avait pris 22 mois de retard, le budget initial (280MF) était dépassé de 40%

- *"50% des projets informatiques dépassent le budget prévu"* (Gartner)

Dérapages



Étude réalisée par The Standish Group International (completée en 2002)

Pourtant, ailleurs. . .

- En génie civil (ponts, autoroutes, tunnels,. . .) c'est "moins pire"

  Ex : Viaduc de Millau, décembre 2004, livré à temps, 310 M€

Différences

- Les caractéristiques des projets informatiques diffèrent de celle des projets de génie civil
  - diversité des applications
  - objectifs exacts non connus avant l'achèvement
  - niveau d'abstraction
  - invisibilité
  - taille
  - complexité
  - coût élevé
  - évolutivité des besoins
  - répartition
  - parallélisme
  - etc. . .

Différences

- Chaque projet informatique est un cas nouveau
  - Il y a peu de références standards.
  - Développer un logiciel s'apparente plus à une activité de recherche qu'à la routine.

- Le cahier des charges n'est presque jamais complet et figé
  - Il s'élabore et s'adapte à mesure de l'avancement du projet parce que les systèmes informatiques sont trop complexes pour être maîtrisés par des êtres humains, et parce que l'informatique est méconnue des décideurs et politiques.

Différences

- Il ne suffit pas de mettre de l'argent et de la technique
  - Plus qu'ailleurs la qualité et la motivation des hommes sont prépondérantes.
  - Un logiciel est un produit industriel atypique, relevant de l'artisanat !

Différences

- Personne ne sait aujourd'hui créer de logiciel sans défaut !
  - La validation de Windows 2000 avait fait appel à 600 000 bêta testeurs, il restait pourtant au lancement de sa commercialisation 63 000 problèmes potentiels dans le code, dont 28 000 sont réels.
  - Un logiciel commercial type contient entre 20 et 30 bugs pour mille lignes !

- Bill JOY, de Sun, a écrit à propos de W2K
  - *"Ce système a atteint un niveau de complexité au delà de toute raison, il est tout simplement impossible de le déboguer."*

Tout n'est pas perdu !

- Il faut s'imposer des processus formels de développement
  - processus d'assurance qualité
    - ▷ **écrivez** ce que vous faites
    - ▷ **faites** ce qui est écrit
    - ▷ **prouvez** que vous le faites
  - documentation pour toutes les phases
    - documents de spécification
    - documents de conception
    - documents de codage
    - documents de recette
    - ...
  - existence de points de contrôle

Tout n'est pas perdu !

- Processus formels...
  - utilisation d'outils de gestion des versions du code source, tels que CVS
  - la méthode doit être structurée, phasée
  - obtenir des produits finis en fin de phase
    - ⇝ inspection et validation après chaque phase du développement
  - être automatisée, adaptable, avec un processus formel et exhaustif de tests
  - utiliser des technologies à jour (objets, Java, AGL, services web, SOA,...)

Tester, tester, tester

- Tests de boîte noire
  - Le test porte sur le fonctionnement externe du système. La façon dont le système réalise les traitements n'entre pas dans le champ du test.
- Tests de boîte blanche
  - Le test vérifie les détails d'implémentation, c'est à dire le comportement interne du logiciel.
- Tests de conformité
  - Le test vérifie la conformité du logiciel par rapport à ses spécifications et sa conception.

Tester, tester, tester

- Tests de non conformité
  - Le test vérifie que les "cas non prévus" ne perturbent pas le fonctionnement du système.
- Tests bêta
  - Réalisés par des développeurs ou des utilisateurs sélectionnés, ils vérifient que le logiciel se comporte pour l'utilisateur final comme prévu par le cahier des charges.
- Tests alpha
  - Le logiciel n'est pas encore entièrement fonctionnel, les testeurs alpha vérifient la pré-version.

Tester, tester, tester

- Tests unitaires
  - Chaque module du logiciel est testé séparément, par rapport à ses spécifications, aux erreurs de logique.
- Tests d'intégration
  - Les modules validés par les test unitaires sont rassemblés. Le test d'intégration vérifie que l'intégration des modules n'a pas altéré leur comportement.
- Tests fonctionnels
  - L'ensemble des fonctionnalités prévues est testé : fiabilité, performance, sécurité, affichages,...

Tester, tester, tester

- Tests d'intégration système
  - L'application doit fonctionner dans son environnement de production, avec les autres applications présentes sur la plate-forme et avec le système d'exploitation.
- Tests de recette
  - Les utilisateurs finaux vérifient sur site que le système répond de manière parfaitement correcte.
- Tests de non régression
  - Après chaque modification, correction ou adaptation du logiciel, il faut vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

Tester, tester, tester

Qualité du logiciel

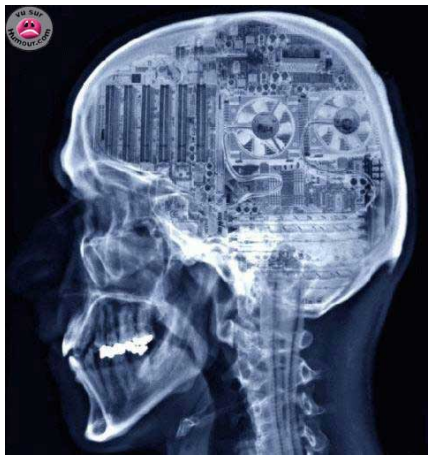
- **Des questions à se poser très en amont**

▷ Portabilité
  - utilisation d'un langage standardisé
  - indépendance du matériel
  - indépendance du système d'exploitation

▷ Utilité
  - fiabilité
    → précision, intégrité
    → tests réussis pour tous les cas, uniformité de conduite
  - efficacité
    → économie de mémoire, rapidité d'exécution
  - aspect humain
    → facilité d'utilisation, accessibilité
    → documentation pour l'utilisateur

Qualité du logiciel

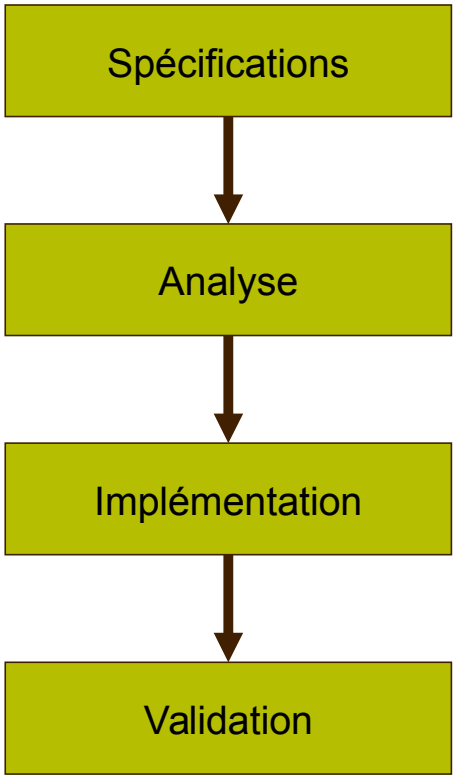
- Maintenabilité
  - facilité de vérification
    - → autodocumentation
    - → vérifications formelles statiques
    - → structuration
  - clarté
    - → structuration
    - → concision
    - → lisibilité
  - facilité d'adaptation
    - → structuration
    - → facilité d'extension
    - → documentation technique



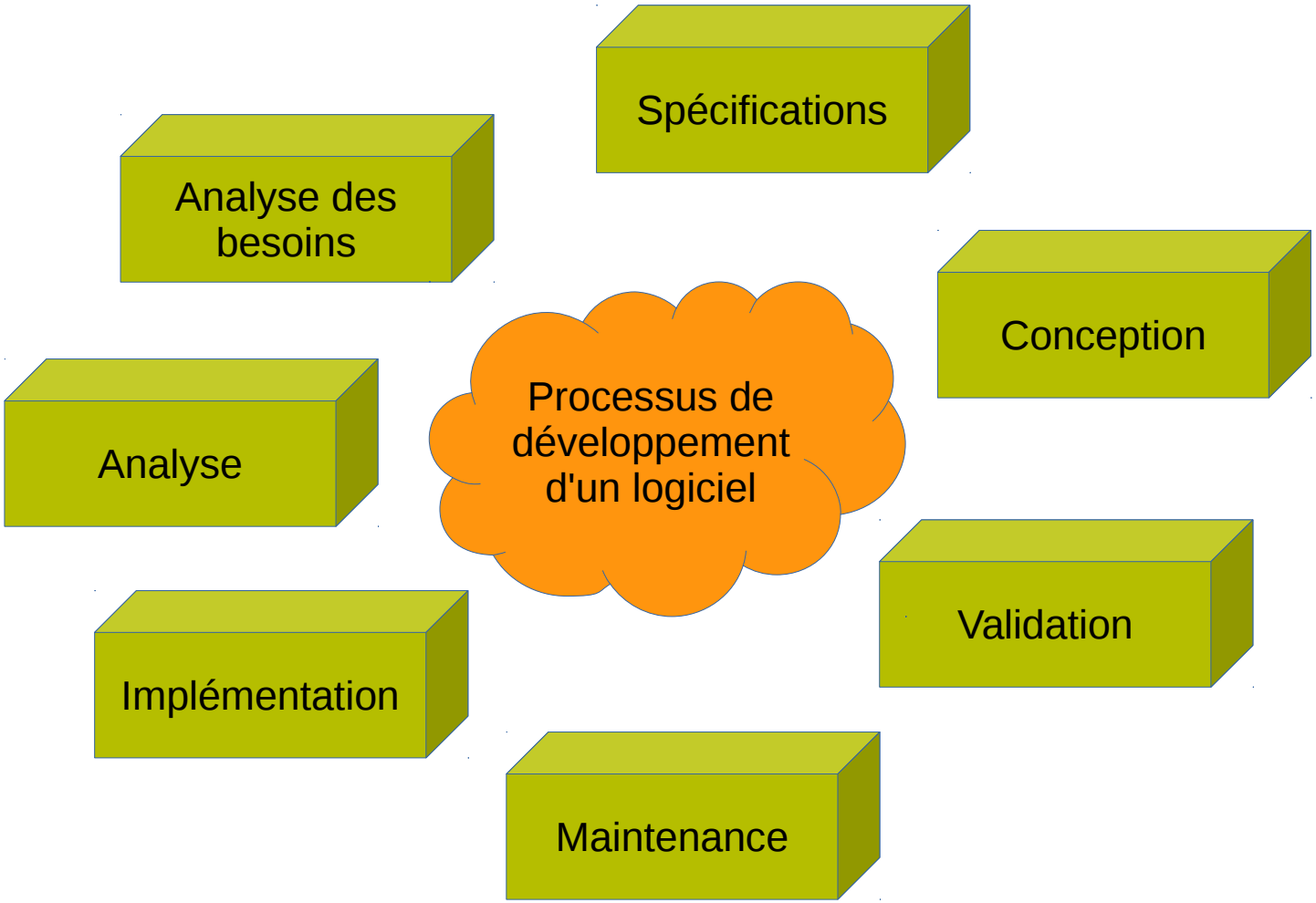
Plan du cours

Programmation "in-the-small"

- Spécifications
  - données, complètes et précises
- Analyse descendante
  - décomposition fonctionnelle
- Implémentation
  - un seul langage de programmation
- Validation
  - jeux d'essai



Étapes du processus de développement



Analyse des besoins

- Étape préalable si le client n'a qu'une idée peu précise du système à réaliser
  - Étude informelle des fonctionnalités (externes) du système sans considération technique : **point de vue métier/utilisateur**
  - Dialogue fournisseur/client en termes intelligibles pour ce dernier : l'aider à **formaliser le problème à résoudre**
  - Produit un document textuel avec schémas...
  - Conduit généralement à la définition d'un **cahier des charges**

Analyse des besoins : exemple

- Contrôle d'accès à l'IUP de Vannes
  (adapté par J.Y.Antoine de P.A.Muller et N.Gaertner, 2000, Modélisation objet avec UML, Eyrolles)
  - Le bâtiment de l'IUP se divise en 4 zones (enseignement TD, enseignement TP, administration et enseignants) correspondant à des droits d'accès différents. 200 personnes ont quotidiennement accès au site. En dehors des visiteurs, toutes les autres catégories de personnes disposent d'une carte d'accès : enseignants (20), personnels administratifs (15) et étudiants (150). Les visiteurs n'auront accès au bâtiment qu'aux heures de libre ouverture (voir planning associé). En dehors de ces horaires, l'autorisation d'accès dans chaque zone sera limitée aux porteurs de badge et dépendra de leur statut (enseignant, administratif, étudiant) et de l'horaire. Chaque zone est contrôlée par une porte d'accès munie d'un lecteur de badge spécifique. Les droits d'accès ainsi que le planning des horaires d'ouverture seront définis par un superviseur. Etc...

Spécifications

- **Ce que doit faire le système** (côté client)
  - ▷ Document précis spécifiant les fonctionnalités attendues
  - ▷ Base du contrat commercial avec le client
  - ▷ Document facile à comprendre par le client/utilisateur

  Ex : Définition de la frontière du système, description des fonctionnalités du système avec scénarios, interactions (enchaînements d'écrans) → cas d'utilisation de Jacobson

- **Les spécifications ne sont jamais complètes et définitives**
  - Évolution du domaine
  - Besoins supplémentaires
  - ...

Spécifications : exemple

- Cas d'utilisation de l'application "accès IUP"
  - use case : configuration système (acteur : superviseur)
  - use case : demande d'accès (acteur : porteur de badge)
  - ...
- Use case : configuration système
  1. identification
     a. saisie du code personnel
     b. vérification du code personnel
     c. autorisation
  2. modification de la configuration
     a. choix d'une zone
     b. ...

Analyse du système

- **"Quoi faire" : comprendre et modéliser le métier**
  - ▷ Réflexion métier hors de toute considération technique
  - ▷ Reste un support de discussion avec le client/utilisateur
  - ▷ Premier modèle du système (niveau métier)
    - identifier les éléments intervenants hors (acteurs) et dans le système : fonctionnalités, structures et relations, états par lesquels ils passent suivant certains événements,...
    - exemples : diagrammes de collaborations, de classes,...

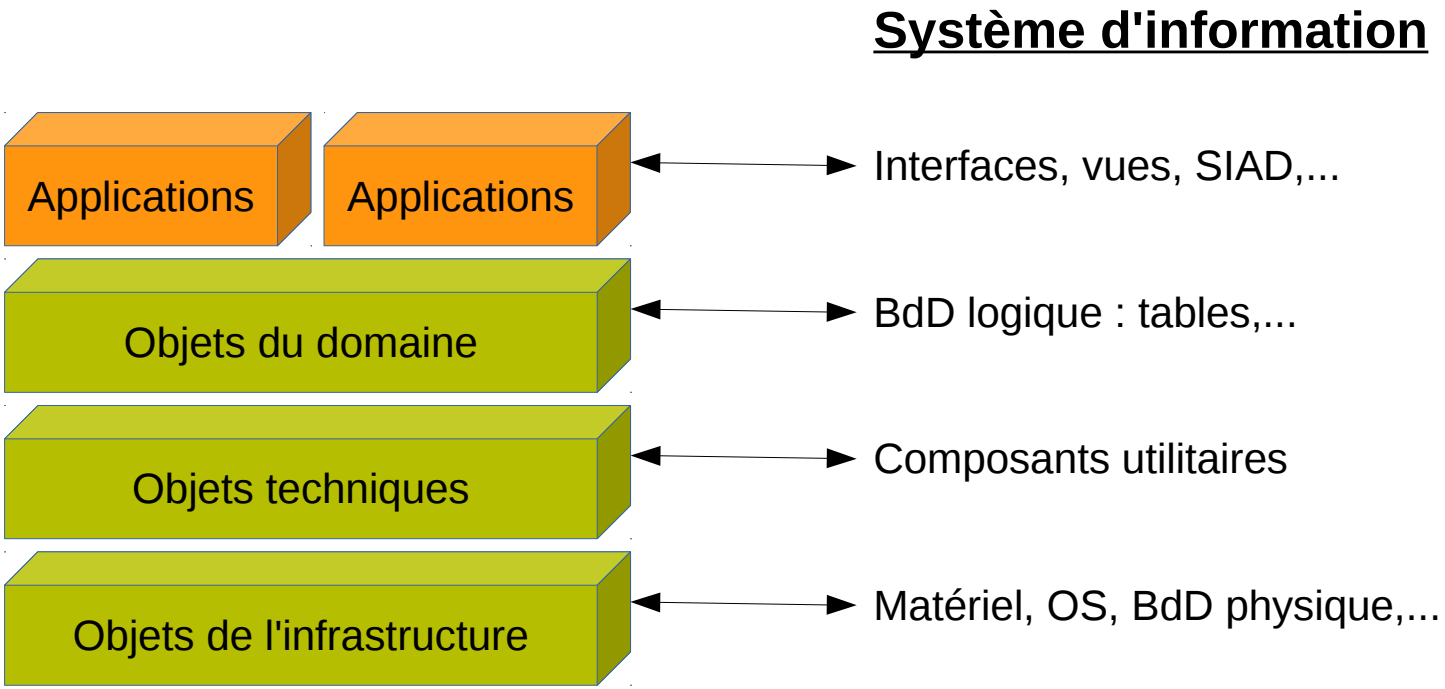
⚠ **L'analyse n'est jamais complète, mais elle doit être juste**

Conception

- Comment faire le système : choix techniques
  - Choix d'une architecture technique (matériel, logiciel) suivant certaines priorités (facteurs qualités : robustesse, efficacité, portabilité,...)
  - Expertise informatique : hors compréhension du client
  - **Modèle d'architecture logicielle du système**
    - décomposition en sous-systèmes : application (interfaces), domaine (métier) et infrastructure (implémentation)
    - permet la définition des phases d'implémentation, de validation et de maintenance

Conception

- Modèle d'architecture en couches



Implémentation

- Souvent trop de temps consacré au codage au détriment des phases d'analyse et de conception $\rightsquigarrow$ mauvaise pratique, parfois très coûteuse...

- Savoir user de la réutilisabilité de composants, voire d'outils de génération de code (mise en place automatique du squelette du code à partir du module système)

$\Rightarrow$ L'activité de développement sera de plus en plus orientée vers la **réutilisation de composants existants**

Maintenance

- Deux types de maintenance
  - ▷ Correction des erreurs du système
  - ▷ Demande d'évolution (modification de l'environnement technique, nouvelles fonctionnalités)
- Facteurs de qualité essentiels
  - Correction $\rightsquigarrow$ robustesse
  - Évolutions $\rightsquigarrow$ modifiabilité, portabilité
- Étape longue, critique et coûteuse
  - 80% de l'effort de certaines entreprises (pb de pratiques ?)

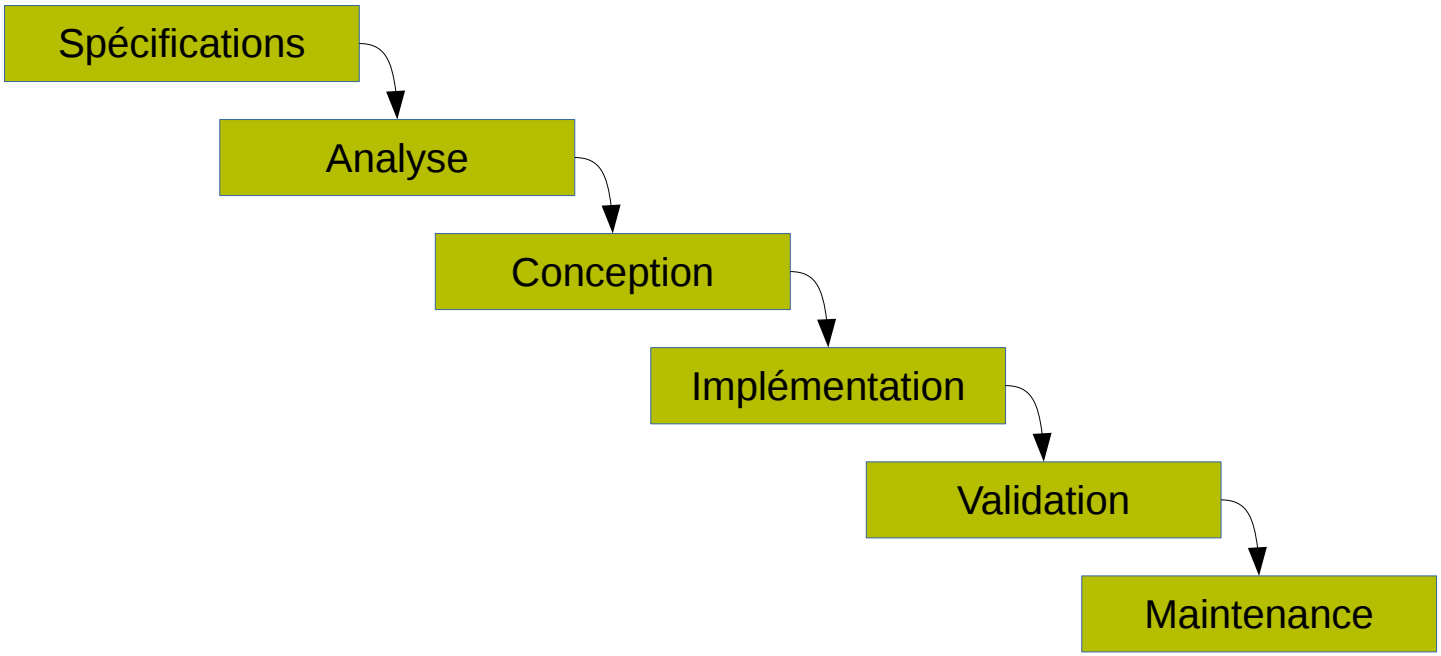
Plan du cours

Cycle de vie d'un logiciel

- Cycle de vie = séquencement des différentes étapes du processus de développement
- Cycles de vie linéaires
  - modèle en cascade
  - modèle en "V"
- Cycle de vie itératif
  - modèle en spirale (ou incrémental) $\rightsquigarrow$ prototypes

Modèle en cascade

- Cycle de vie linéaire sans aucune évaluation entre le début du projet et la validation



Modèle en cascade

- Inconvénient : aucune validation intermédiaire !
  - impossibilité de suivre le déroulement du projet, donc de planifier un travail en équipe
  - augmentation des risques car validation tardive : erreur d'analyse ou de conception très coûteuse !
    - → *le coût de correction d'une erreur est multiplié par 10 à chaque phase*
    - → *on parle aussi du "Big Bang"* ☺

⇒ Solution limitée aux petits projets

- ▷ risques bien délimités dès le début du projet
- ▷ projet court avec peu de participants

Théorie vs réalité

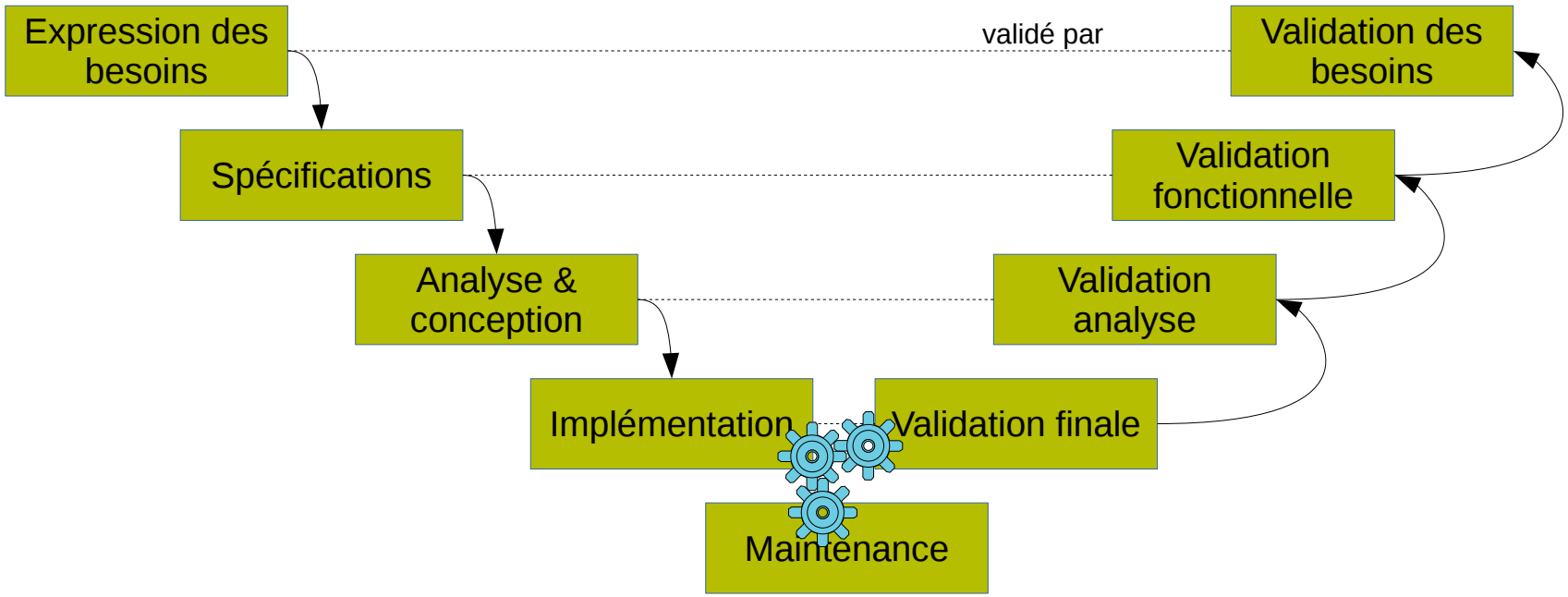
- L'informatique a créé beaucoup de "lois" pour décrire le décalage entre la théorie et la réalité
  - **Loi de Brooker** : dix grammes d'abstraction valent des tonnes de bricolage
  - **Loi de Klipstein** : les défauts n'apparaissent qu'après que le système ait passé (avec succès) la phase d'intégration
  - **Loi de Brook** : doubler le nombre de programmeurs sur un projet en retard ne fait que doubler le retard
  - **Loi de Weinberg** : si les architectes construisaient les maisons comme les programmeurs écrivent les programmes, le premier pivert venu détruirait la civilisation
  - **Loi de Myers** : on passe la moitié de son temps à refaire ce que l'on n'a pas eu le temps de faire correctement

Cycle en "V"

- Objectif
  - mieux gérer le risque et faciliter la planification
- Principes :
  - ▷ processus linéaire
  - ▷ prévention des erreurs → validation 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

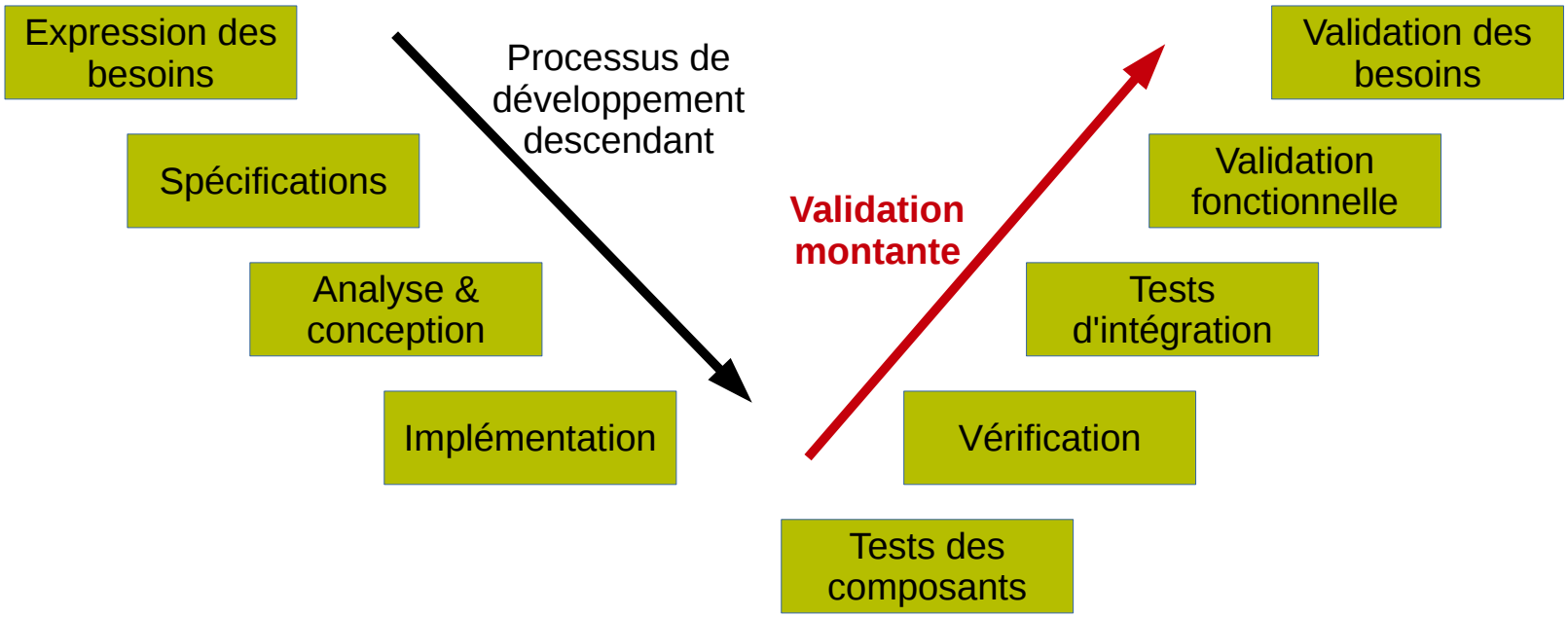
Cycle en "V"

- Analyse descendante
  - validations intermédiaires "papier" → documentation



Cycle en "V"

- Validation
  - protocoles de validation définis par l'analyse descendante



Cycle en "V"

- Intérêts :
  - validations intermédiaires
    - bon suivi de projet : points de mesure concrets de l'avancement du travail avec étapes clés
    - favorise la décomposition fonctionnelle de l'activité
    - limitation des risques en cascade par validation de chaque étape
    - existence d'outils support
  - modèle éprouvé très utilisé pour de grands projets
  - pourtant...

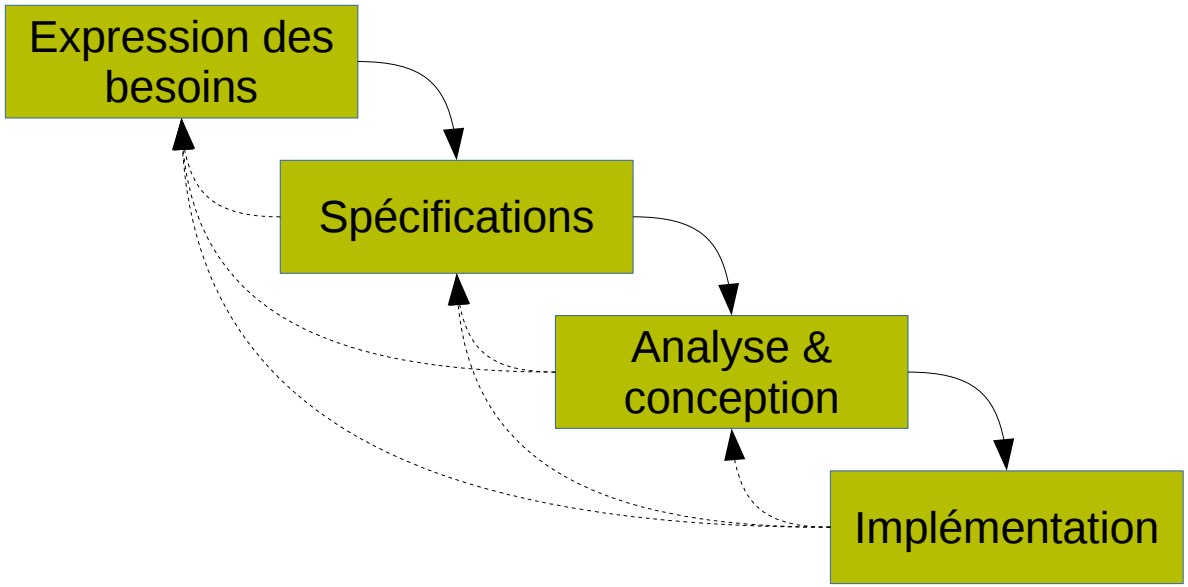
Cycle en "V"

- 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 zone d'ombre
    - idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires

Améliorations du cycle en "V"

- Retours de correction dans les phases précédentes
  - réponse ad hoc pour casser la linéarité du processus
  - fonctionne si les besoins de correction sont limités

  ⇝ cycle de vie itératif



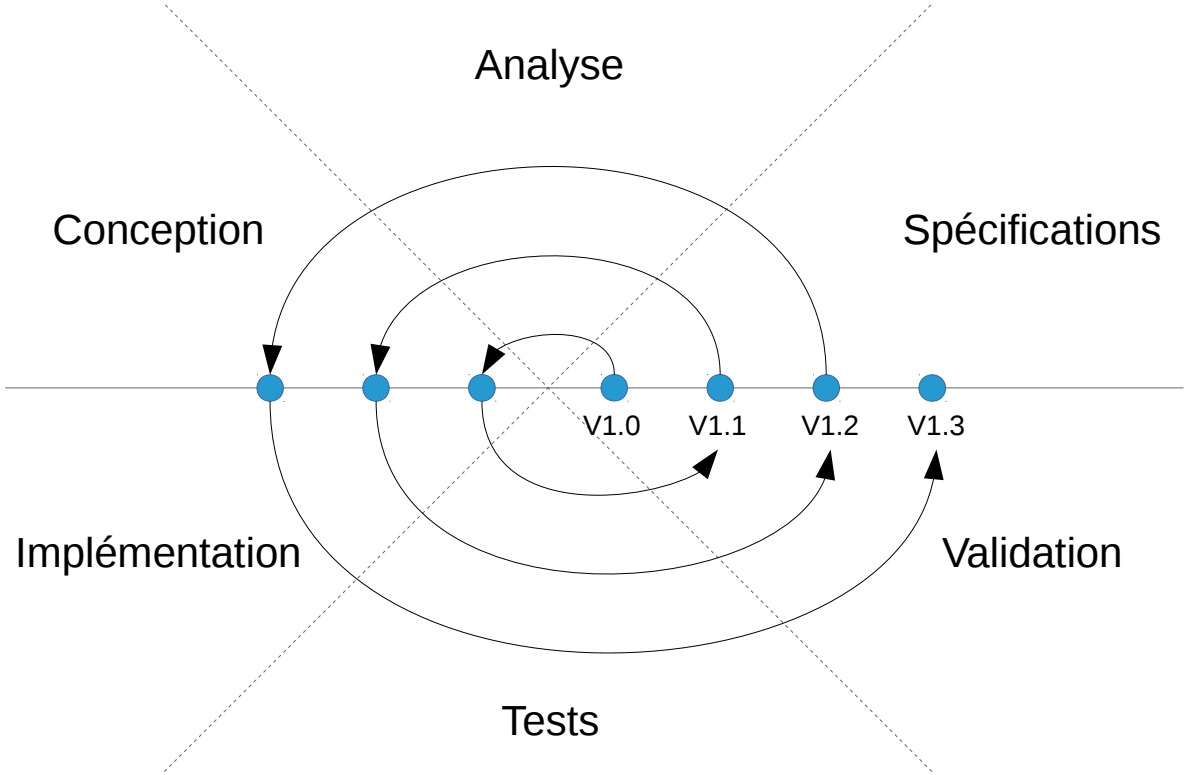
Cycle de vie itératif

- Prototypage
  - idée : fournir le plus rapidement possible un prototype exécutable permettant une validation concrète et non sur document
  - progression du projet par incréments successifs de versions successives du prototype : **itérations**
  - certains prototypes peuvent être montrés aux clients et utilisateurs → une **maquette** peut être réalisée préalablement au 1er prototype
  - la validation par prototypes ne justifie pas l'absence de recours à la **documentation** !

Cycle de vie itératif

- Itération = mini cycle de vie en cascade

⇝ Spirale du cycle itératif



Cycle de vie itératif

- Criticité de la 1ère itération
  - ne pas chercher à réaliser le système complet à la 1ère itération ⇒ travers d'un processus linéaire
- Intérêts du cycle itératif :
  - ▷ validation concrète et non sur documents
  - ▷ limitation du risque à chaque itération
  - ▷ 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 ça passe"
  - ▷ flexibilité
    - modification des spécifications ⇝ nouvelle itération
    - maintenance ⇝ forme particulière d'itération
  - ▷ planification renforcée

Cycle de vie itératif

- Fausses limitations :
  - n'encourage pas la bidouille
  - ne crée pas de problèmes
    → *au contraire il les révèle*
  - ne demande pas de refaire la roue à chaque itération
    → *enrichissement et non recommencement*
  - n'encourage pas la modification constante des besoins
    → *hiérarchise avant tout les besoins*

Cycle de vie itératif

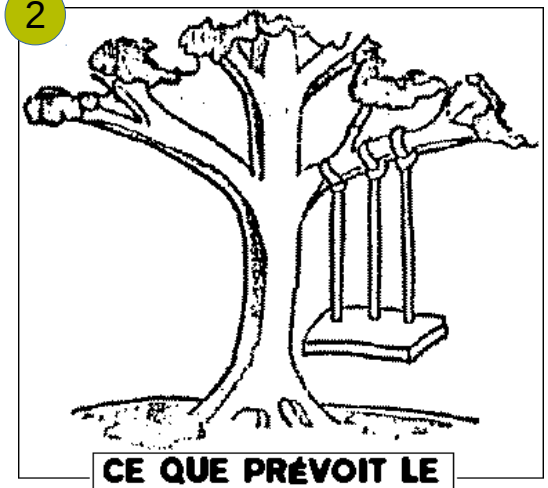
- Limitations :
  - pas de processus idéal
    - *cycle itératif : planification très attentive et rigoureuse*
    - *cycle en "V" : processus éprouvé le plus répandu, surtout pour les systèmes connus (le cycle itératif peut dérouter au 1er abord)*
  - processus surtout adapté à la modélisation objet
    - *le modèle objet se prête parfaitement à une démarche incrémentale*
    - *le cycle en spirale a cependant une portée générale*

Il faut **valider** !

1

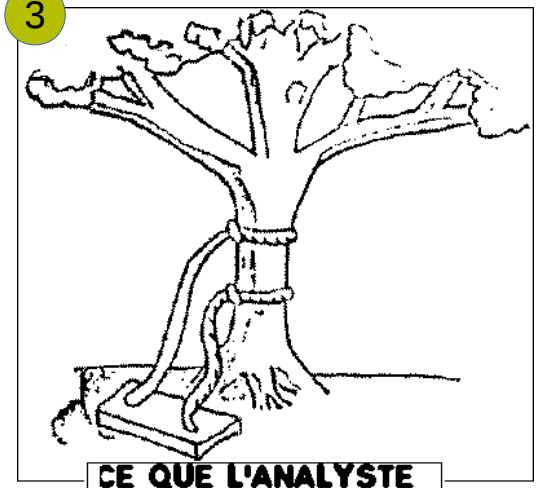
CE QUE DEMANDE LE CLIENT

2



CE QUE PRÉVOIT LE CONTRAT

3



CE QUE L'ANALYSTE A PRÉVU

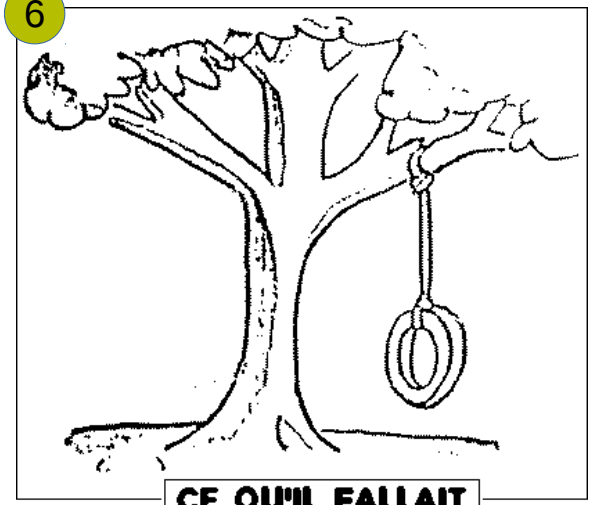
4

CE QUE LE PROGRAMMEUR A ÉCRIT

5

CE QUE LA MISE AU POINT A FAIT

6



CE QU'IL FALLAIT

Plan du cours

Introduction

- Bibliographie
  - Grady Booch, James Rumbaugh, Ivar Jacobson (2000). *Le guide de l'utilisateur UML*.
  - Bertrand Meyer (2000). *Conception et programmation orientées objet*.
  - G.Masini, A.Napoli, D.Colnet, D.Léonard, K.Tombre (1997). *Les langages à objets, langages de classes, langages de frames, langages d'acteurs*.

Programmation orientée objet

Définition

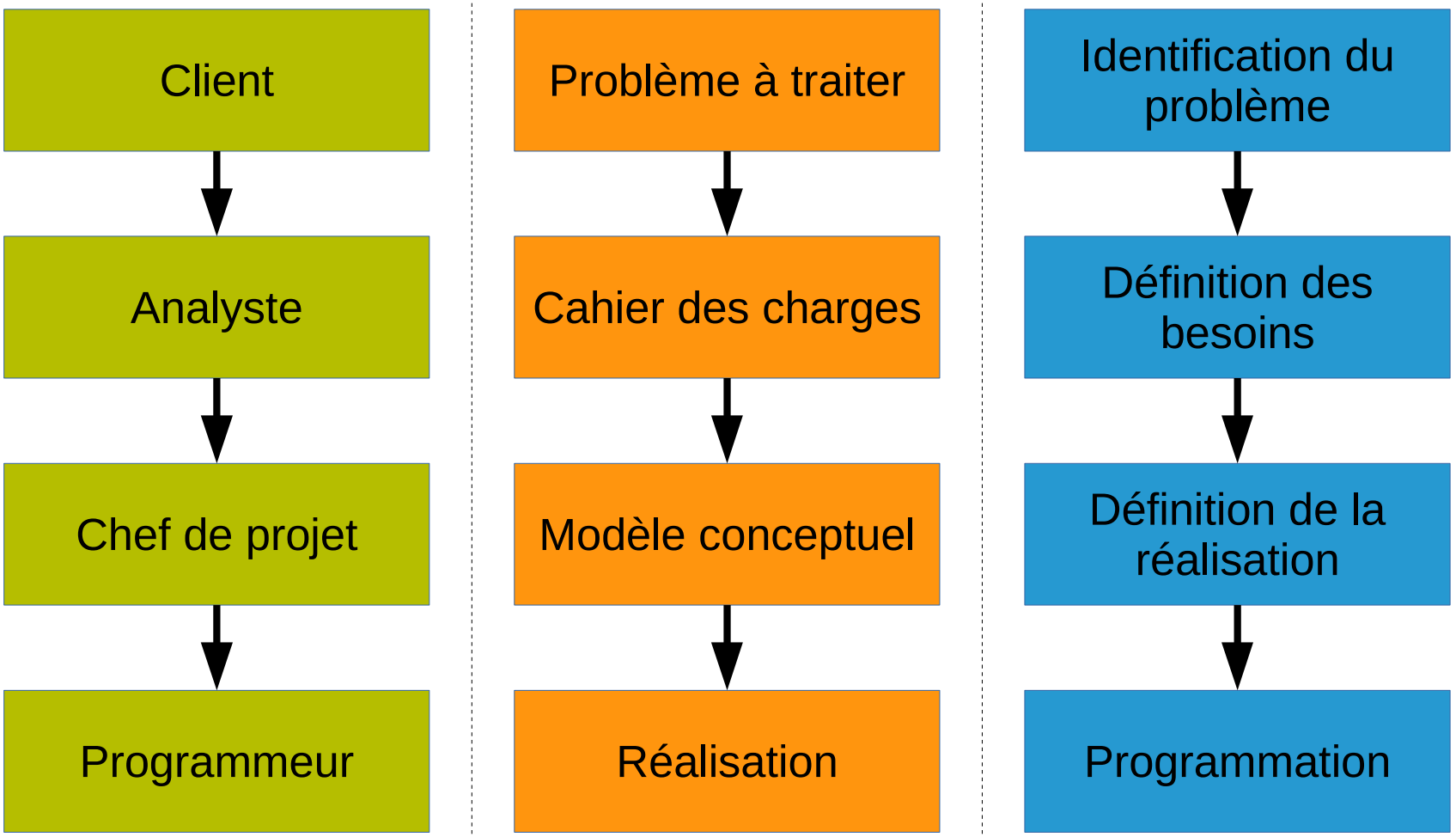
Façon d'architecturer une application informatique en regroupant les données et les traitements sur ces dernières au sein d'une même entité : les objets.

- Objectif
  - organiser, permettre les évolutions
- Méthode
  - encapsulation des données et des traitements associés

Comment répondre à un problème ?

1. Définir et analyser le problème
2. Conceptualiser le problème
3. Proposer une solution
4. Réaliser la solution

Schéma classique



Concepts de base

- **Objet**
  - Entité de base regroupant un ensemble de caractéristiques (données ou procédures)
- **Encapsulation**
  - Les détails de l'implémentation d'un objet sont cachés aux autres objets du système. Cette action consiste alors à donner des points d'entrée dans l'objet pour en connaître son état.
- **Modularité**
  - Permet de diviser le programme pour en réduire la complexité.

Concepts de base

- **Héritage**
  - Permet de partager des caractéristiques entre deux objets. Ainsi, tous les objets héritant d'une même super-classe possèdent tous les mêmes caractéristiques définies par cette dernière.

- **Polymorphisme**
  - Permet de donner différentes implémentations de la même caractéristique. L'héritage permet justement de donner un polymorphisme à un objet.

Comment trouver un objet ?

- Choisir ses caractéristiques/attributs
- Choisir ses comportements/méthodes
- Méthodologie de désagrégation/agrégation
  - décomposer le "tout" en un ensemble de "parties", chaque partie devenant à son tour un tout
  - approche ≠ décomposition fonctionnelle

Quelques règles

- Un module représente **un** concept et **tout** le concept
- Ne représenter que ce qui existe
- Ne pas créer de module "fourre-tout"
- Bien choisir le nom du module en fonction de ce qu'il exprime (cf. Design Patterns)
- Astuce : les méthodes correspondent à des verbes (→ actions)

Design patterns

1977 : travaux de Christopher Alexander

Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois sans jamais le faire deux fois de la même manière.

- Bibliographie
  - "gang of four" = Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*.

Design patterns

- Patron ou motif de conception
  - formalisation de bonnes pratiques
- Destiné à résoudre les problèmes récurrents
  - décrit une solution standard pour des problèmes d'architecture
  - pattern ≠ algorithme

  ⇒ ne pas devoir réinventer la roue à chaque fois !
- Notion issue de la POO

Design patterns

- Classés en plusieurs catégories
  - **Création**
    - comment faire l'instanciation et la configuration des classes et des objets

    ex fabrique, fabrique abstraite, singleton,...
  - **Structure**
    - comment organiser les classe d'un programme

    ex adaptateur, façade, proxy, objet composite,...
  - **Comportement**
    - comment organiser les objets pour que ceux-ci collaborent

    ex commande, itérateur, observateur, médiateur,...

  ⋮

Design patterns

- La description d'un patron de conception est faite selon un formalisme fixe :
  - Nom
  - Description du problème à résoudre
  - Description de la solution
    - les éléments (objets) de la solution, avec leurs relations (en UML)
    - la solution est appelée motif de conception
  - Conséquences → résultats issus de la solution

Plan du cours

Introduction

- Les méthodes objets
  - 1970 à 1990 → mise au point des approches OO
  - 1994 → plus de 50 méthodes objet !
- Émergence de 3 méthodes :
  - OMT de Rumbaugh (Object Modeling Technique)
  - BOOCH'93 de Booch
  - OOSE de Jacobson (Object Oriented Software Engineering)

Introduction

- Historique :
  - 1994 v0.8 : Rumbaugh et Booch (OMT et Booch'93) → "Unified Method"
  - 1995 v0.9 : Jacobson (OOSE)
  - 1997 v1.1 : l'OMG standardise l'**Unified Modeling Language**
  - Aujourd'hui : v2.0 (norme ISO)
- Site web : http://www.uml.org

UML

- Caractéristiques :
  - UML = **langage**, ~~méthode de conception~~
  - But : uniformiser la conception d'une application
  - Indépendant de la programmation
  - Notation graphique simple et standardisée
    → ensemble de diagrammes décrivant un projet
  - Format d'échange : XMI (*XML Metadata Interchange*)

9 principaux diagrammes UML

- Diagrammes structurels (vue statique) :
  ⇒ *paquetages, structures composites*
  - cas d'utilisation
  - classes
  - objets
  - composants
  - déploiement
- Diagrammes comportementaux (vue dynamique) :
  ⇒ *communications, interactions globales, temps*
  - séquence
  - activités
  - collaboration
  - états-transitions

Diagrammes vs étapes du processus de conception

- Découverte des besoins
  - diagramme des cas d'utilisation
    - *décrit les fonctions du système selon le point de vue de ses futurs utilisateurs (Jacobson)*
  - diagramme de séquence
    - *représentation des interactions temporelles entre objets (i.e. scenarii)*

Diagrammes vs étapes du processus de conception

- Analyse
  - diagramme de classes
    - *structure des données du système définies comme un ensemble de relations entre classes*
  - diagramme d'objets
    - *illustration des objets et de leurs relations*
  - diagramme de collaboration
    - *représentation des interactions entre objets*
  - diagramme d'états-transitions
    - *comportement des objets d'une classe en termes d'états et de transitions d'états*
  - diagramme d'activités
    - *structure d'une opération en actions*

Diagrammes vs étapes du processus de conception

- Conception
  - diagramme de séquence
    - *représentation des interactions temporelles entre objets dans la réalisation d'une opération*
  - diagramme de déploiement
    - *description du déploiement des composants sur les dispositifs matériels*
  - diagramme de composants
    - *architecture des composants physiques d'une application*

Méthode MM ☺

1. **Spécifier** les classes
   - cas d'utilisation
   - diagrammes de séquence
   - diagramme de classes
2. **Organiser** les composants du système
   - diagramme d'objets
   - diagrammes de collaboration
   - paquetages
   - diagrammes de composants
   - diagramme de déploiement
3. **Dynamique** des objets et des activités
   - diagrammes états-transitions
   - diagrammes d'activités

Cas d'utilisation

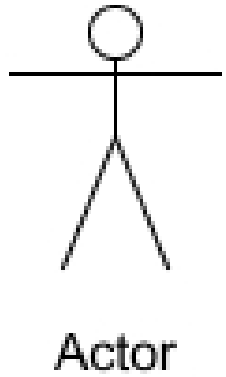
- Déterminer les actions (fonctionnalités) du système
  - ▷ comment va-t-on se servir du système ?
  - ▷ quels sont les acteurs du système ?
  - ▷ quelles sont les actions réalisées par le système ?
- Méthode
  - → définir des scénarii primaires
  - → ne pas traiter les exceptions du système

Ex Retirer de l'argent dans un distributeur

- insérer sa carte de retrait
- taper son code
- choisir le montant
- reprendre sa carte
- récupérer les billets

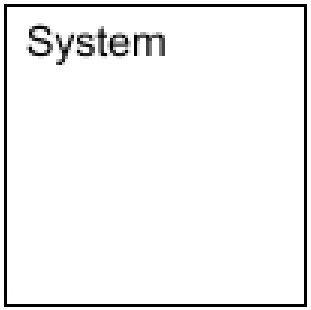
Cas d'utilisation

- Acteur
  - toute entité extérieure au système ayant une action sur le système
  - un acteur peut être une personne ou un autre système
  - une personne peut jouer le rôle de différents acteurs

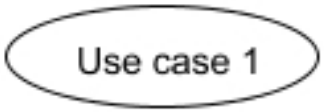


Cas d'utilisation

- Système
  - produit, projet, application que l'on veut écrire
  - contient les actions disponibles pour les acteurs

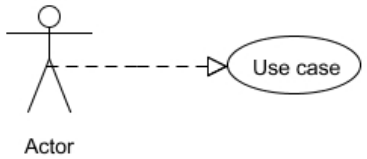


- Use Case
  - action réalisée par l'acteur sur le système

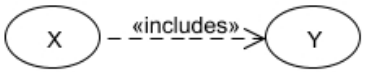


Cas d'utilisation

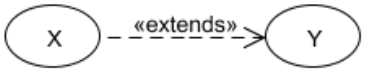
- Relations
  - acteur / use case



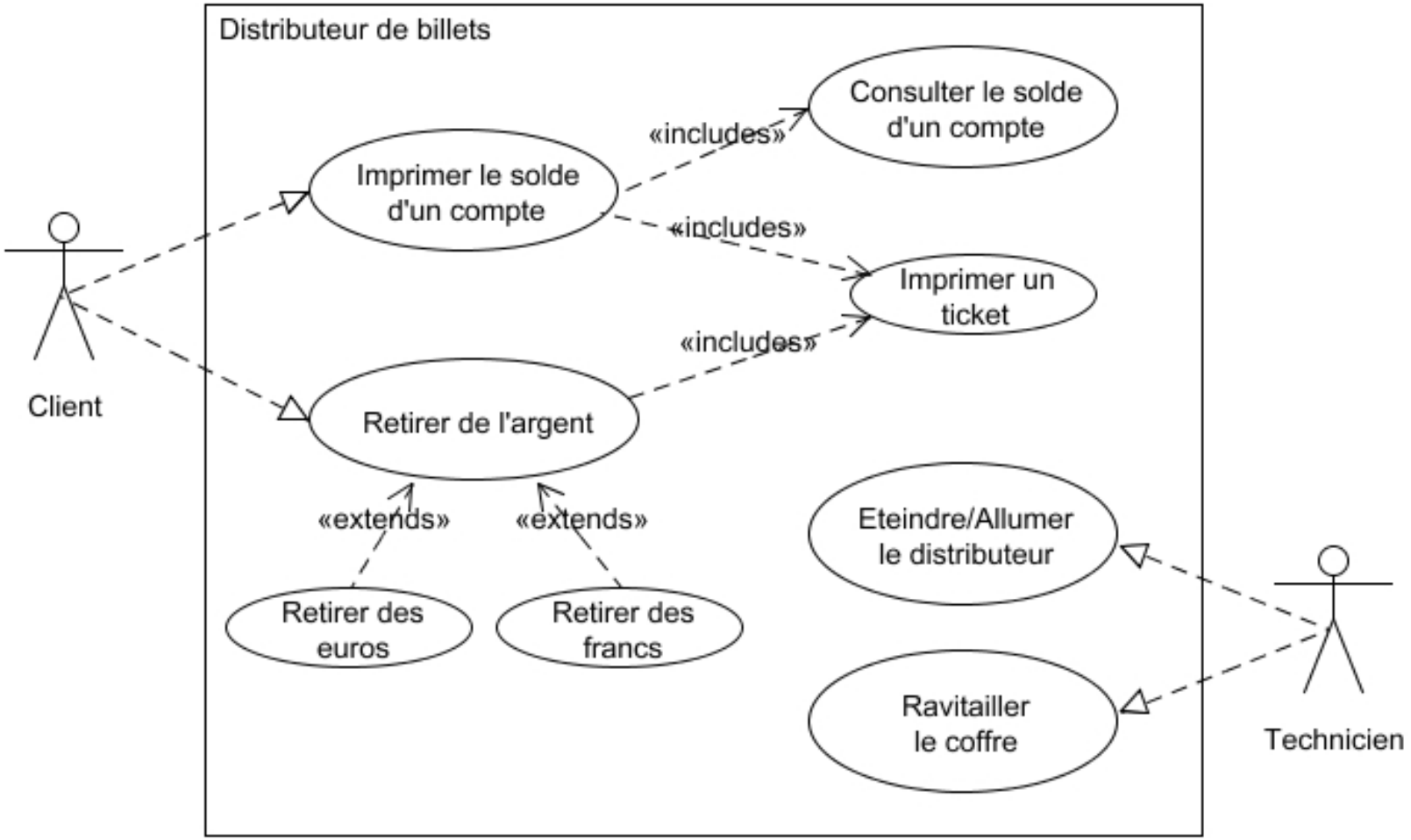
  - inclusion
    - → X contient le comportement de Y



  - extension
    - → X est un cas précisé de Y



Cas d'utilisation : exemple



Cas d'utilisation vs scénario

- 1 système = ensemble de cas d'utilisation
- Le système contient les cas d'utilisation **mais pas les acteurs** !
- 1 cas d'utilisation = ensemble de "chemins d'exécution" possibles
- 1 scénario = 1 chemin particulier d'exécution = 1 séquence d'événements

$\Rightarrow$ 1 scénario = instance d'un cas d'utilisation

Cas d'utilisation vs scénario

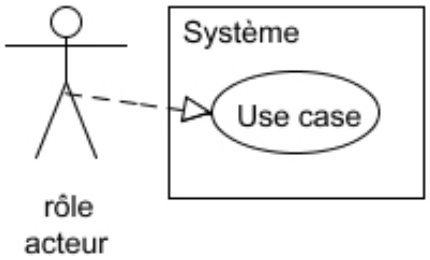


diagramme de cas d'utilisation

$\rightsquigarrow$

instance

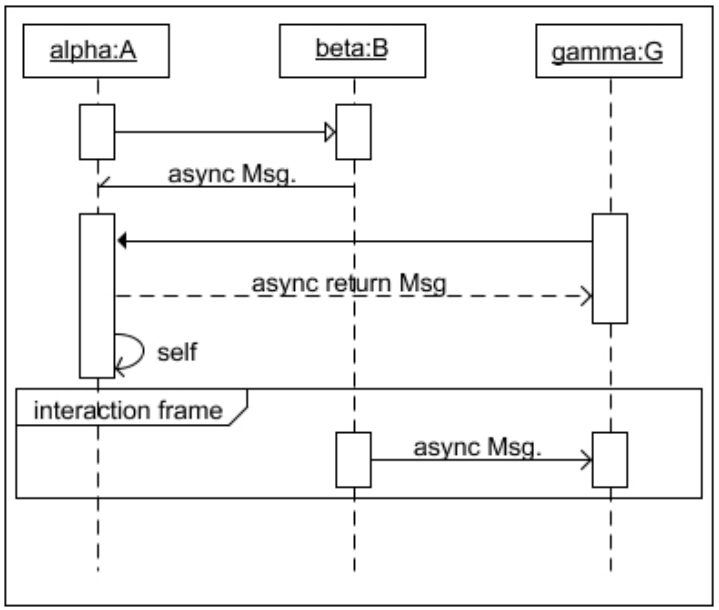


diagramme de séquence = scénario

Scénarii

- La spécification exhaustive de tous les scénarii est difficile, voire impossible

⇒ Sélection des scénarii les plus intéressants

- ▷ scénario optimal
  - *décrit l'interaction la plus fréquente*
- ▷ scénarii dérivés
  - *décrivent certaines alternatives importantes non représentées dans le scénario optimal*

Scénarii

- 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 des instances d'acteurs ou d'objets du système
  - aspect temporel : flot ordonné d'événements

NB On peut également représenter un scénario par un diagramme de collaboration

Diagramme de séquence : exemple

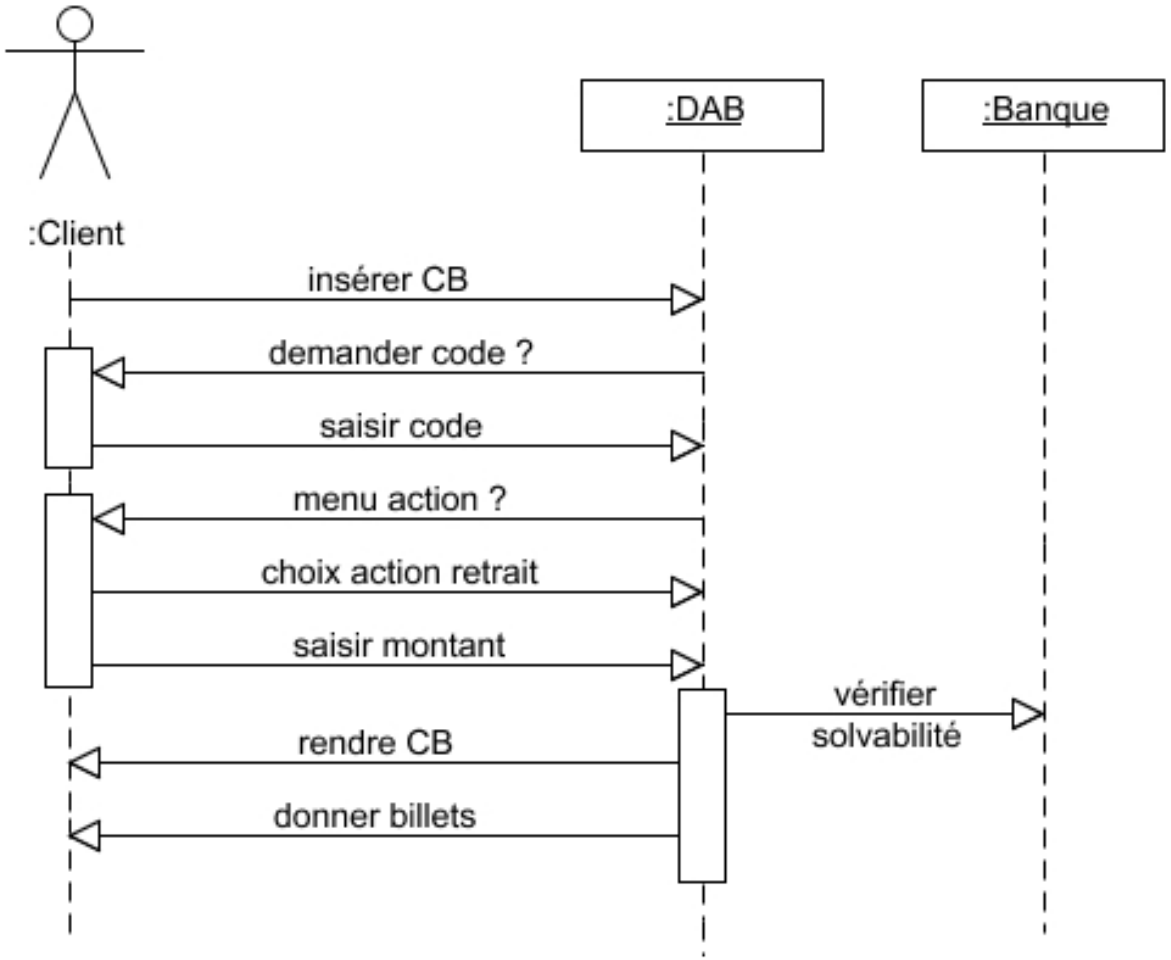
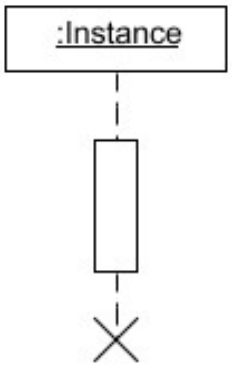
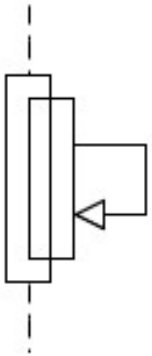


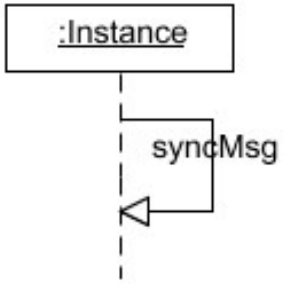
Diagramme de séquence



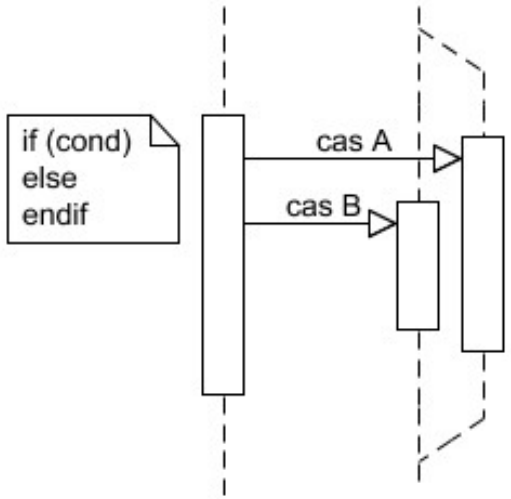
- le temps s'écoule de haut en bas
- les bandes d'activation indiquent l'état d'activation de l'objet (durée d'une méthode)



- exemple de récursivité



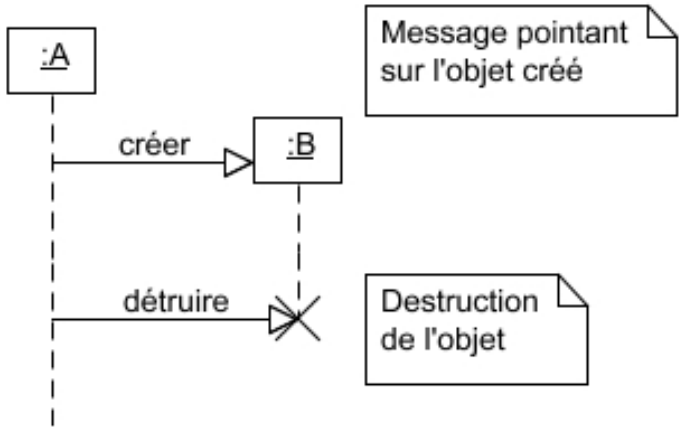
message réflexif



branchement conditionnel

Diagramme de séquence

- Création et destruction d'un objet par message



- Message réflexif

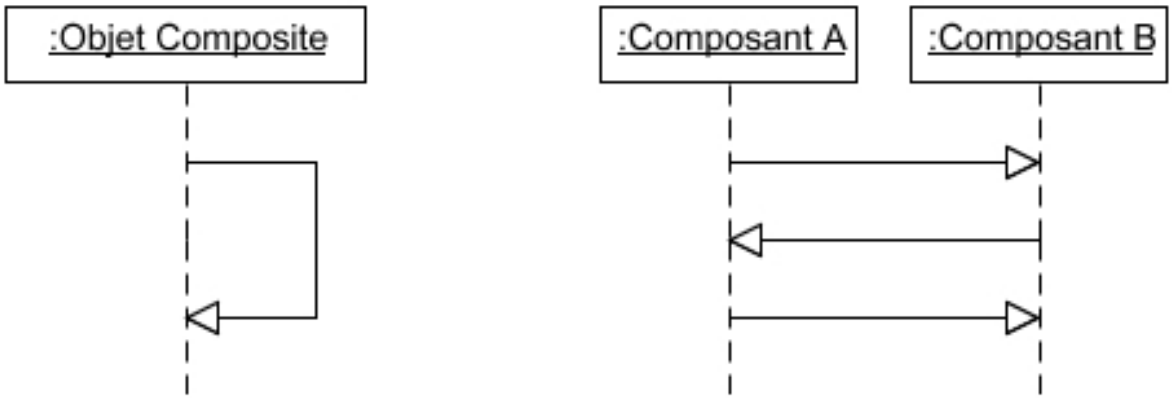
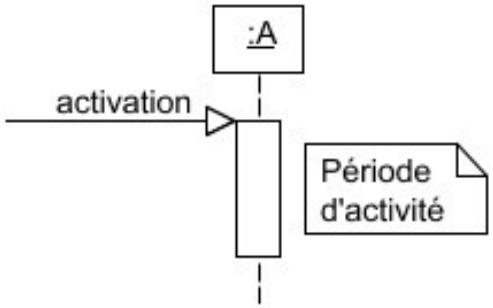
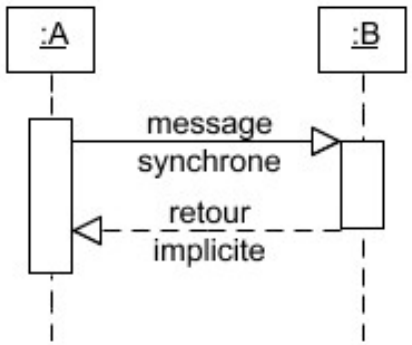


Diagramme de séquence

- Période d'activité des objets



- Messages synchrones vs asynchrones



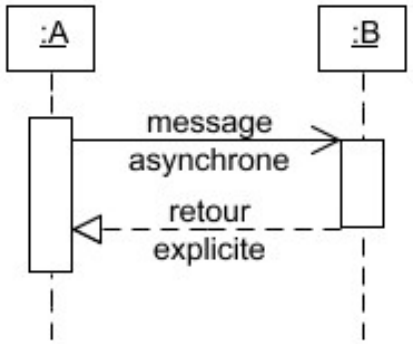


Diagramme de séquence

- La syntaxe UML pour les diagrammes de séquence est bien plus riche
  - ⇝ messages minutés (synchrones avec délai d'attente)
  - ⇝ messages dérobant (asynchrones et déclenchent une action chez le récepteur s'il attendait le message → trap)
  - ⇝ contraintes temporelles
  - ⋮

Diagramme de séquence → diagramme de classes

- Les scénarii (ou diagrammes de séquence) font apparaître :
  - les différents composants du système → les objets et leurs classes
  - leur structure → objets composites, héritage
  - leurs interactions → appels de méthodes
  - une certaine organisation des objets → paquetages ou modules
  - l'enchaînement des actions → "ébauche" ou "squelette" des méthodes
  - ⋮

Diagramme de classes

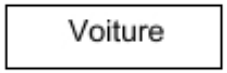
- Nomenclature
  - Classe
    - modèle de l'objet
    - type de données abstrait regroupant des attributs et des méthodes
  - Objet
    - instance d'une classe
    - entité possédant une identité
    - attributs caractérisant son état
    - méthodes définissant son comportement

Diagramme de classes

- Notions complémentaires
  - Classe abstraite
    - classe non instanciable car elle ne définit que des comportements et des attributs servant de base pour d'autres classes
  - Interface
    - classe permettant de simplifier la vue d'une autre classe au reste du système
  - Héritage multiple
    - une classe peut hériter de une ou plusieurs classes
  - Visibilité
    - \+ publique : accès depuis tte entité interne ou externe à la classe
    - − privée : accès à partir des méthodes de la classe uniquement
    - # protégée : accès à partir de la classe ou des sous-classes

Diagramme de classes

- Classe non documentée



- Classe détaillée
  - attributs
  - méthodes
  - protections
  - ⋮

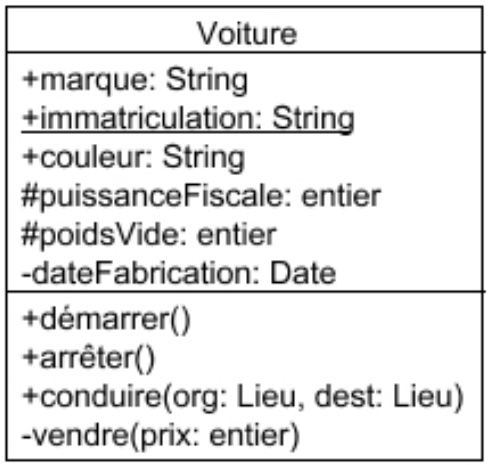
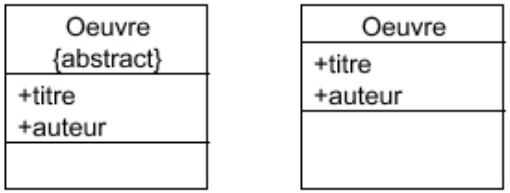


Diagramme de classes

- Classe abstraite
  (notations équivalentes)



- Classe paramétrable
  (template)

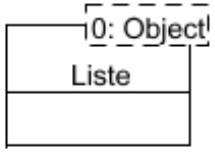
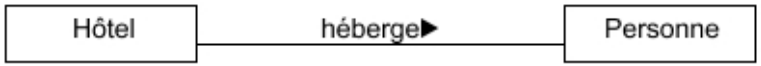
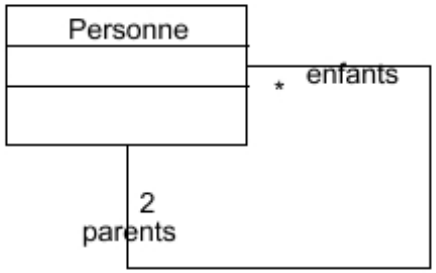


Diagramme de classes

- Relations
  - association
  - association verbale active
  - rôles





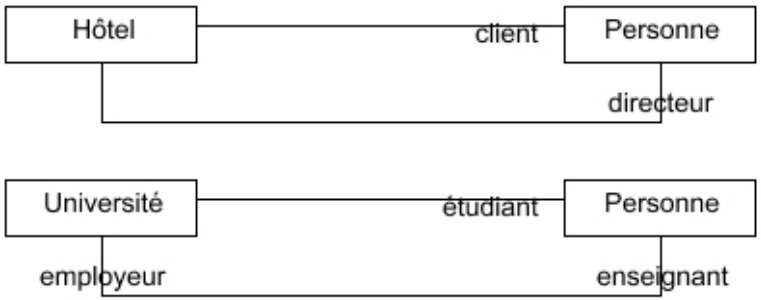
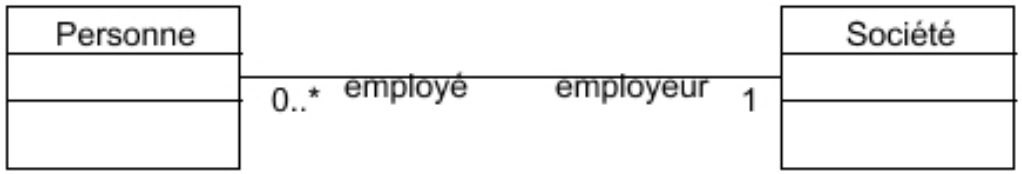


Diagramme de classes

- Cardinalités



1 chaque personne travaille pour une et une seule société (toutes les personnes ont un emploi)

0..* une société emploie de zéro à plusieurs personnes

1	un et un seul
0..1	zéro ou un
* ou 0..*	de zéro à plusieurs (parfois noté 0..n par abus)
1..*	de un à plusieurs (au moins un)
n	exactement n (entier naturel)
m..n	de m à n (entiers naturels)

Diagramme de classes

- Relation de dépendance
  - relation d'utilisation unidirectionnelle et d'obsolescence (une modification de l'élément dont on dépend, peut nécessiter une mise à jour de l'élément dépendant)

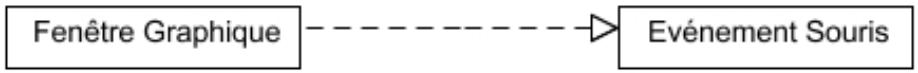
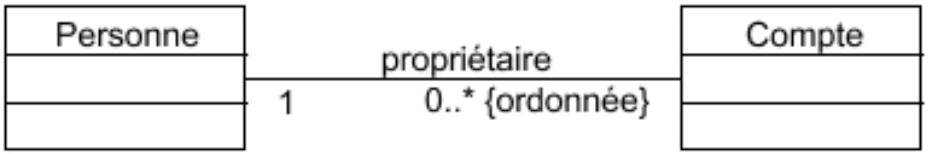


Diagramme de classes

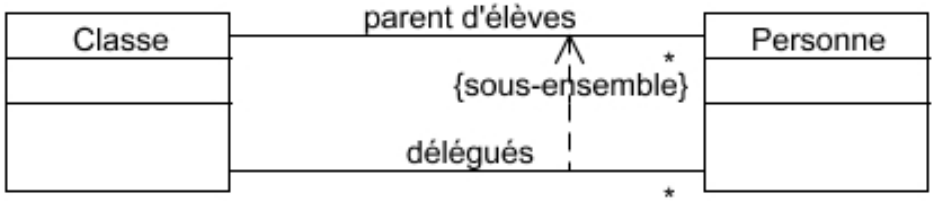
- Contraintes sur les associations



  - la contrainte porte sur une relation ou sur un groupe de relations (notée `{contrainte}`)
  - par exemple, placée sur un rôle, la contrainte `{ordonnée}` définit une relation d'ordre entre les objets de la collection (les comptes) qui sont liés à une personne

Diagramme de classes

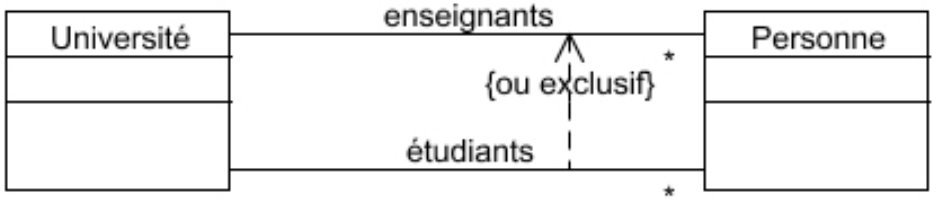
- Contraintes sur les associations



  - la contrainte `{sous-ensemble}` indique qu'une collection est incluse dans une autre collection

Diagramme de classes

- Contraintes sur les associations



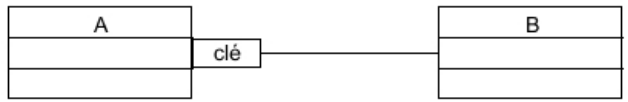
  - la contrainte `{ou exclusif}` précise que, pour un objet donné, une seule association parmi un groupe d'associations est valide

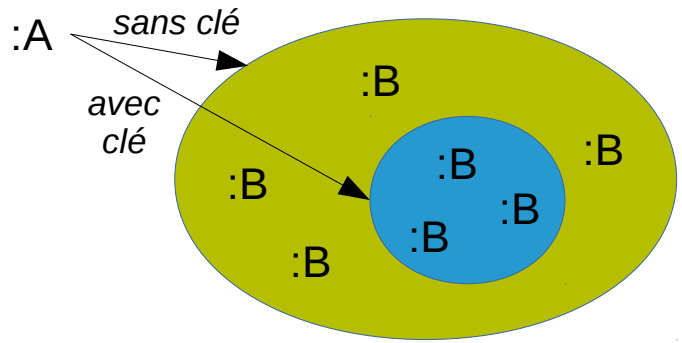
Diagramme de classes

- Restriction des associations
  - la restriction (dite qualification en UML) d'une association consiste à sélectionner un sous-ensemble d'objets parmi l'ensemble des objets qui participent à une association
  - elle est réalisée au moyen d'une clé, ensemble d'attributs particuliers
  - la clé appartient à l'association et non aux classes associées

Diagramme de classes

- Restriction des associations
  - chaque instance de la classe A, accompagnée de la valeur de la clé, identifie un sous ensemble des instances de B qui participent à l'association





  - autre exemple

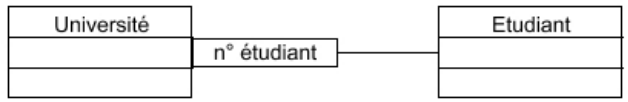


Diagramme de classes

- Association particulière : agrégation
  - une agrégation est une association non symétrique → l'une des extrémités joue un rôle prédominant par rapport à l'autre
  - elle se justifie dans les cas suivants :
    - une classe B "fait partie" intégrante d'une classe A
    - les valeurs d'attributs de la classe A se propagent dans les valeurs d'attributs de la classe B
    - une action sur la classe A implique une action sur la classe B
    - les objets de la classe B sont subordonnés à ceux de la classe A

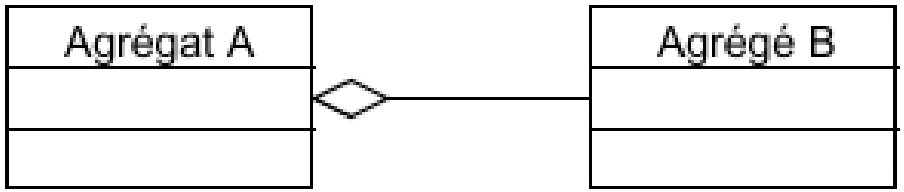


Diagramme de classes

- Agrégation
  - l'agrégation peut être multiple, comme une association classique

  - en tant que "propriétaire", une personne est un agrégat d'immeubles...
  - les immeubles dont elle est propriétaire font partie de la description d'une personne

Diagramme de classes

- Agrégation particulière : composition
  - la composition est une forme particulière d'agrégation
  - le composant est "physiquement" contenu dans l'agrégat → la durée de vie du composant est limitée à celle de l'agrégat
  - la composition implique une contrainte sur la valeur de la multiplicité du côté de l'agrégat : 0 ou 1
  - la valeur 0 du côté de l'agrégat permet un attribut non renseigné

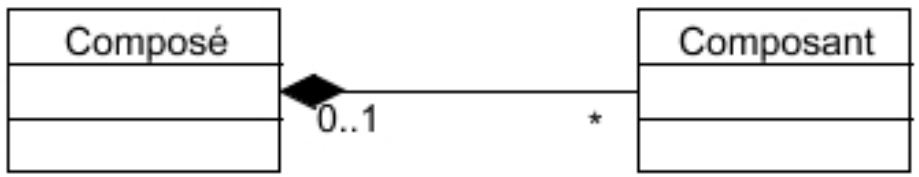


Diagramme de classes

- Remarque : association n-aire
  - on se limite généralement aux associations binaires

Ex une association ternaire entre les classes `Salle`, `Enseignant` et `Classe` peut être réifiée comme une classe `Cours` ayant deux attributs `début` et `fin`

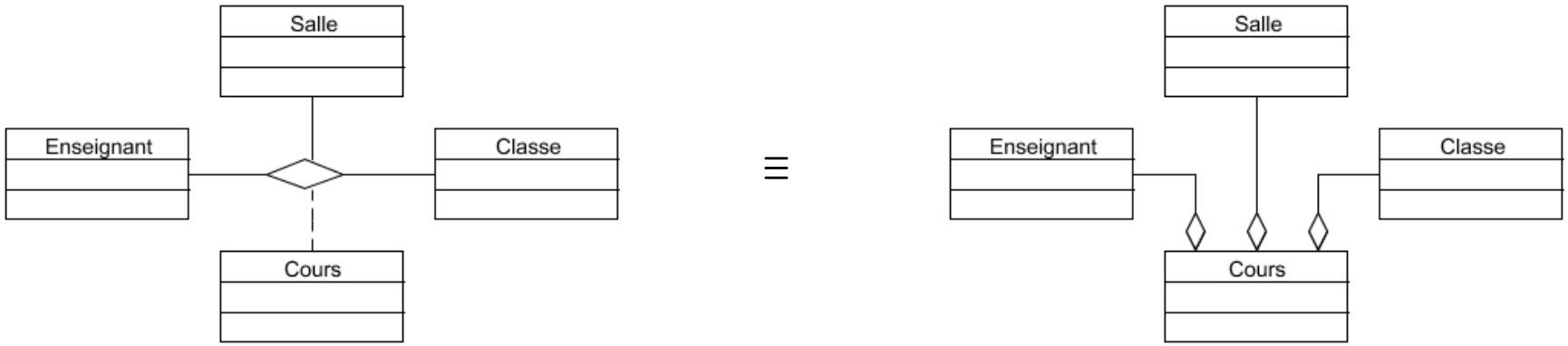


Diagramme de classes

- Association, agrégation, composition

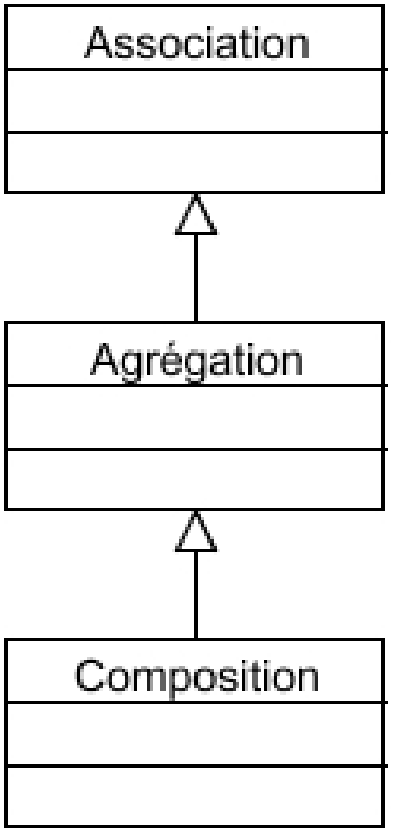
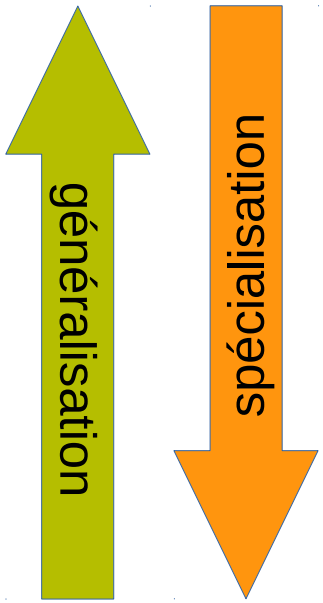
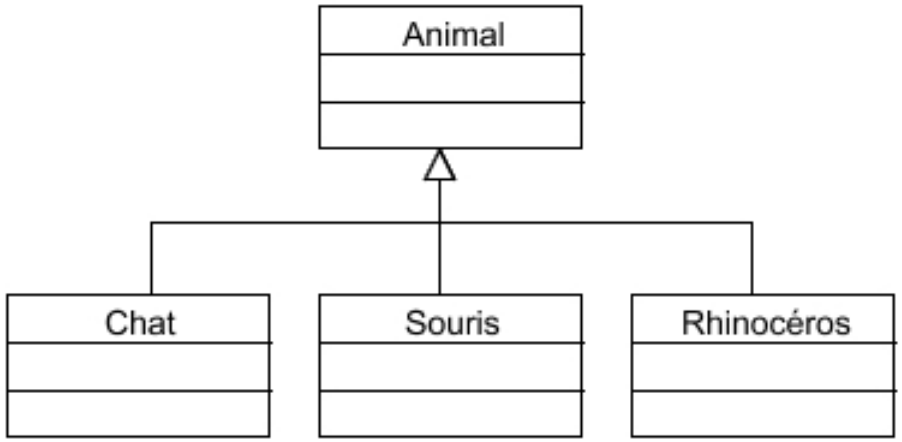


Diagramme de classes

- Généralisation & spécialisation
  - la relation de généralisation signifie "est de" ou "est une sorte de" → notion d'héritage





héritage simple : chaque sous-classe n'a qu'une seule classe mère

Diagramme de classes

- Héritage multiple

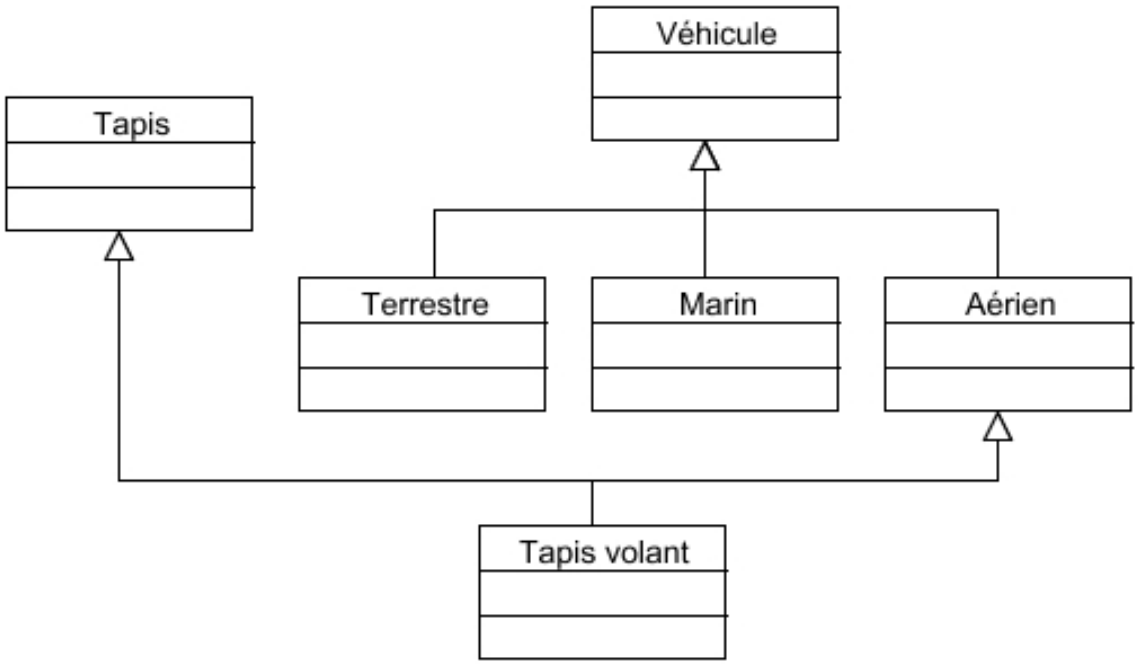


Diagramme de classes

- Contraintes de généralisation
  - une classe peut être spécialisée selon plusieurs critères
  - certaines contraintes peuvent être posées sur les relations de généralisation
  - par défaut, la généralisation symbolise une décomposition exclusive

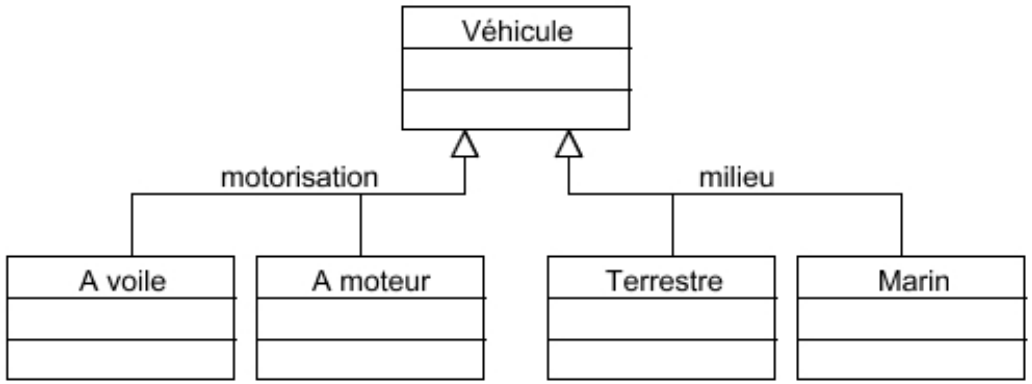


Diagramme de classes

- Contraintes de généralisation
  - la contrainte `{disjoint}` (ou `{exclusif}`) indique la participation exclusive d'un objet à l'une des collections spécialisées

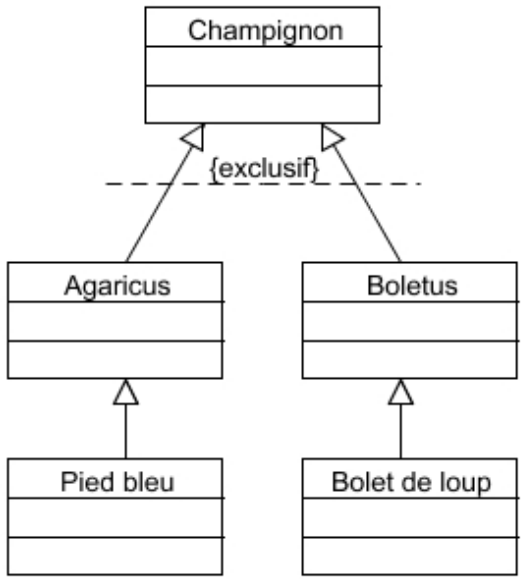


Diagramme de classes

- Contraintes de généralisation
  - la contrainte `{chevauchement}` (ou `{inclusif}`) indique la participation possible d'un objet à plusieurs collections spécialisées

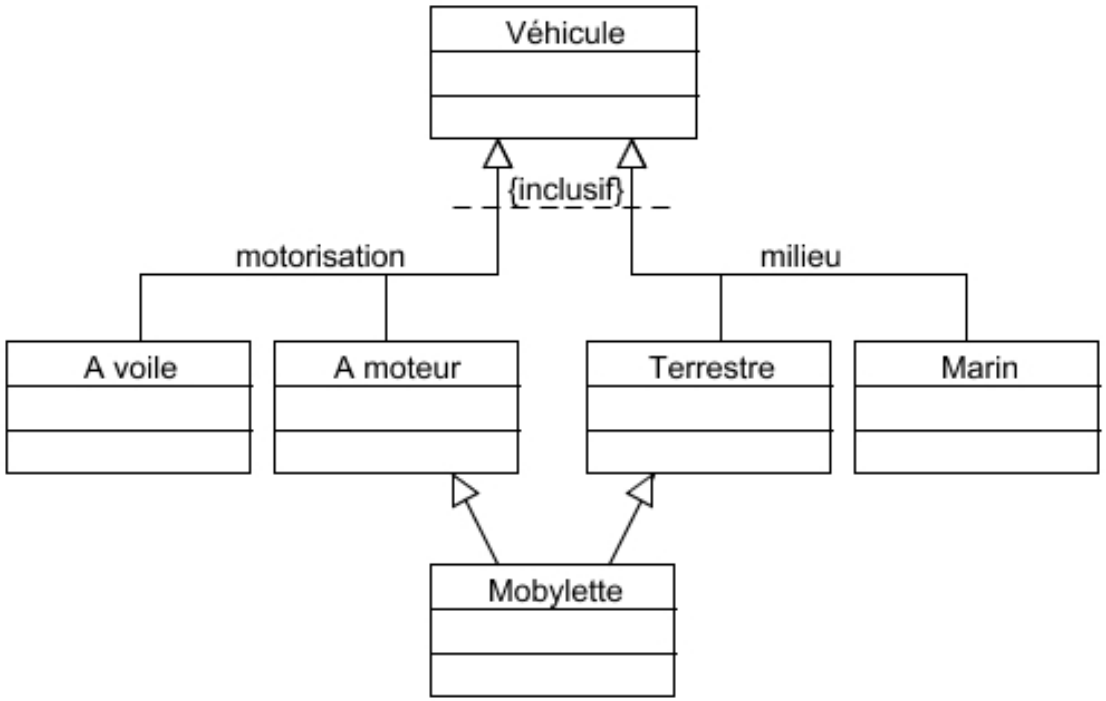
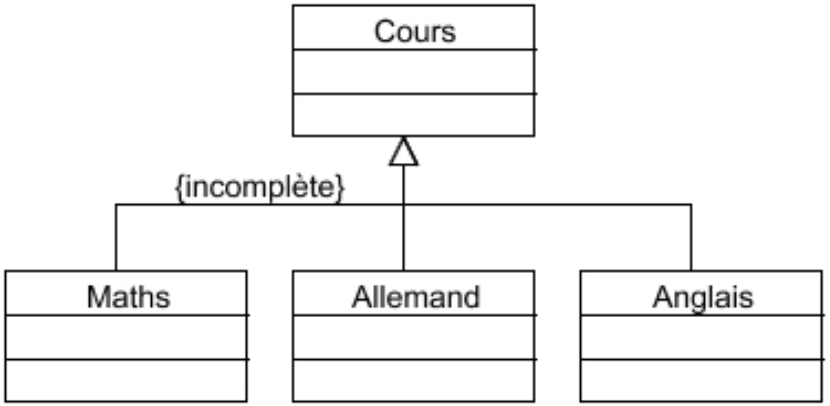


Diagramme de classes

- Contraintes de généralisation
  - la contrainte `{complète}` indique que la généralisation est terminée : tout ajout de sous-classe est alors impossible
  - à l'inverse, la contrainte `{incomplète}` indique une généralisation extensible



Méthode MM ☺

1. **Spécifier** les classes
   - cas d'utilisation
   - diagrammes de séquence
   - diagramme de classes
2. **Organiser** les composants du système
   - diagramme d'objets
   - diagrammes de collaboration
   - paquetages
   - diagrammes de composants
   - diagramme de déploiement
3. **Dynamique** des objets et des activités
   - diagrammes états-transitions
   - diagrammes d'activités

Diagramme de classes vs diagramme d'objets

- Diagramme de classes
  - contient des classes, associations, rôles, cardinalités, dépendances
  - montre l'architecture
- Diagramme d'objets
  - contient des objets (des instances de classes) et leurs liens structurels
  - facilite la compréhension de structures complexes
  - montre un contexte (avant/après une interaction par exemple)
  - utile dans la phase exploratoire

Diagramme d'objets

- 3 représentations possibles des instances
  - instance anonyme

    :Voiture

  - instance nommée

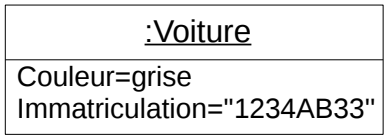
    twingo:Voiture

  - instance nommée d'une classe anonyme

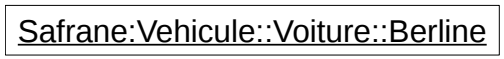
    twingo

Diagramme d'objets

- Instance anonyme dont la valeur de certains attributs est spécifiée



- Instance nommée d'une classe dont on spécifie le chemin complet



- Collection d'instances anonymes de la classe `Voiture`

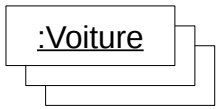


Diagramme d'objets

- Les valeurs des attributs sont optionnelles ainsi que les liens entre objets

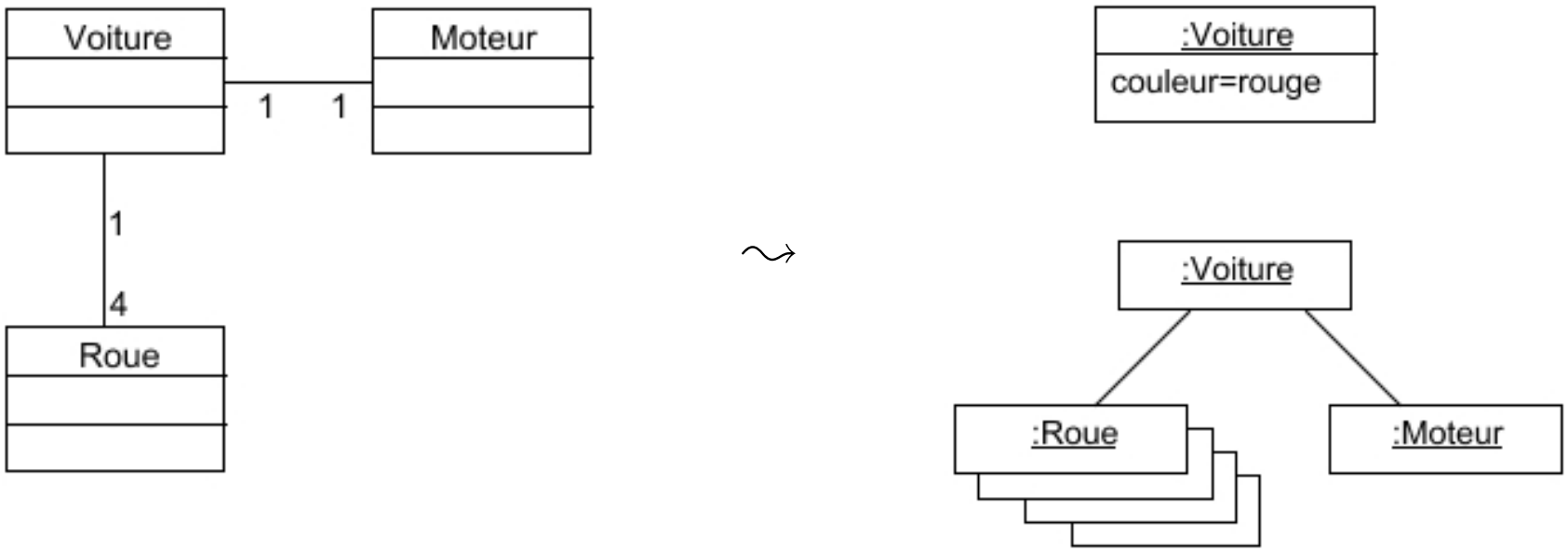


Diagramme d'objets

- Les diagrammes d'objets facilitent la compréhension et l'élaboration d'un diagramme de classes

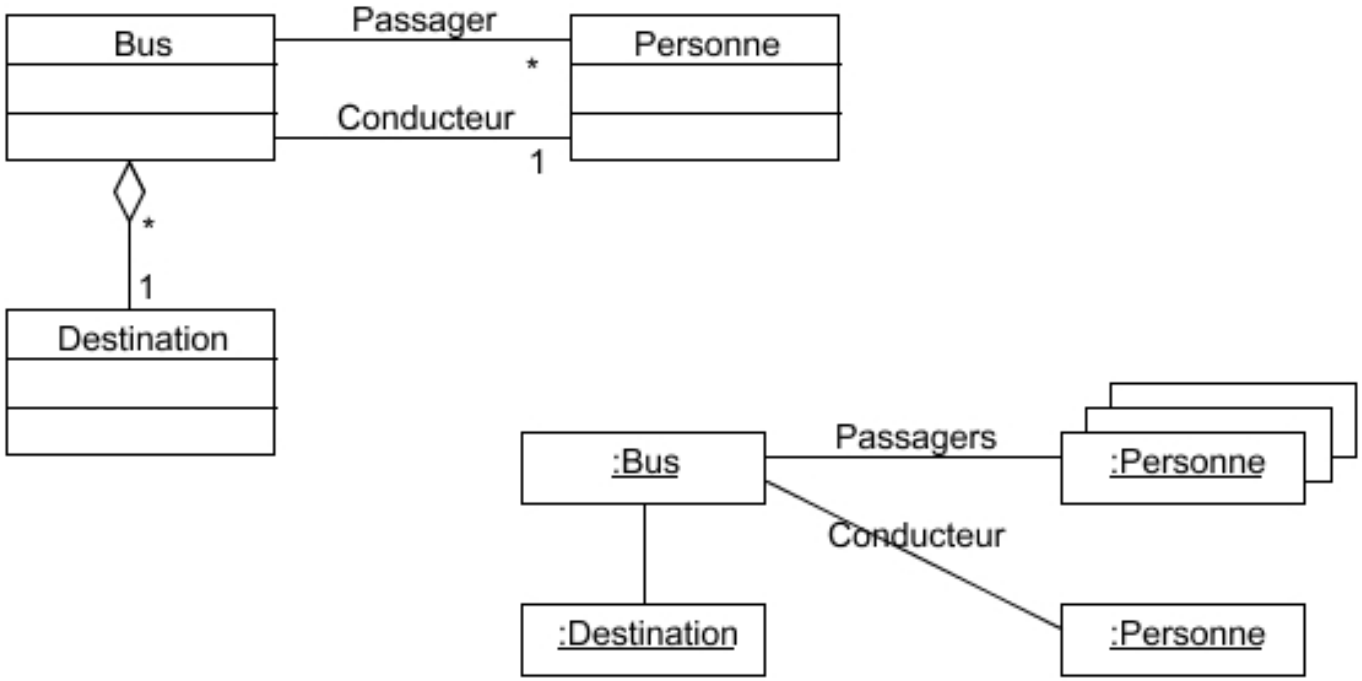
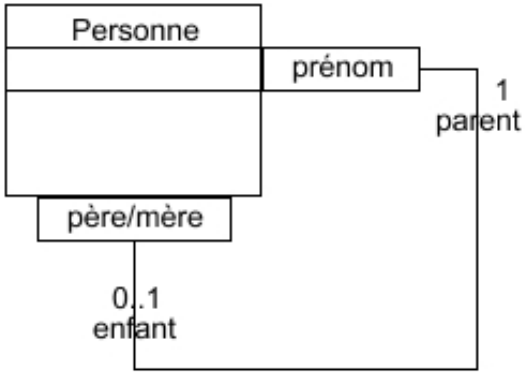
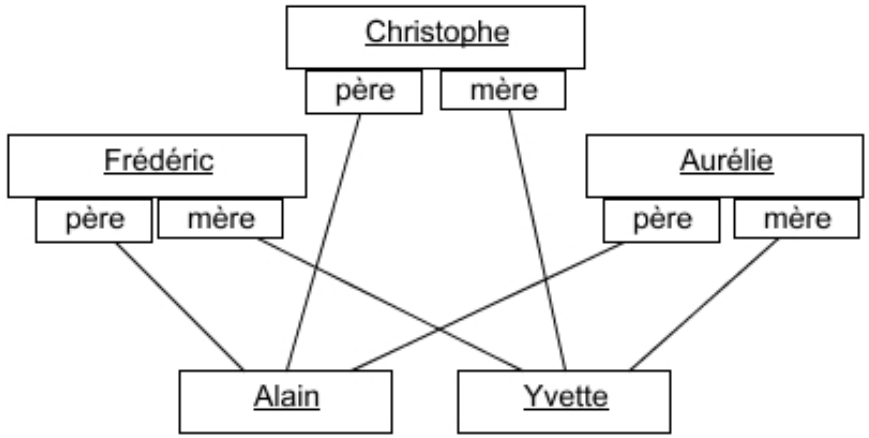


Diagramme d'objets

- Structures complexes





Méthode MM ☺

1. **Spécifier** les classes
   - cas d'utilisation
   - diagrammes de séquence
   - diagramme de classes
2. **Organiser** les composants du système
   - diagramme d'objets
   - diagrammes de collaboration
   - paquetages
   - diagrammes de composants
   - diagramme de déploiement
3. **Dynamique** des objets et des activités
   - diagrammes états-transitions
   - diagrammes d'activités

Diagramme de collaboration

- Extension des diagrammes d'objets → vue dynamique
  - ▷ décrit le comportement collectif d'un ensemble d'objets
  - ▷ en vue de réaliser une opération
  - ▷ en décrivant leurs interactions modélisées par des envois (éventuellement numérotés) de messages

- Dualité diagramme de collaboration/diagramme de séquence
  - diagramme de séquence
    - ⇝ aspects temporels des interactions
  - diagramme de collaboration
    - ⇝ qui communique avec qui ?
    - ⇝ nœuds "centraux" du système

Diagramme de collaboration

- Message = `nom_opération`

- Envois éventuellement numérotés → ordre des envois de message au cours d'une opération

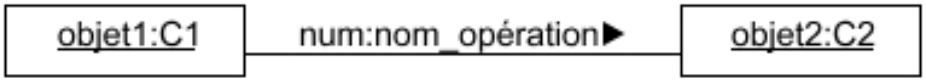
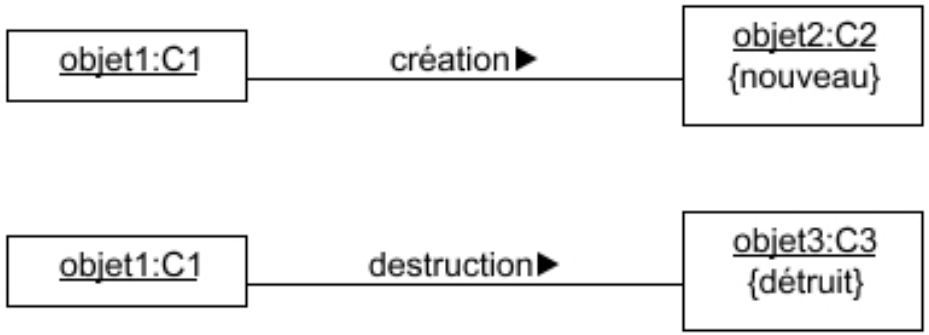


Diagramme de collaboration

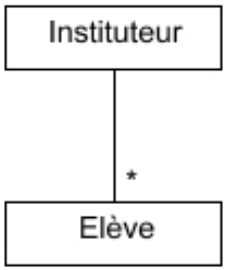
- Les objets (et les liens) créés ou détruits au cours d'une interaction peuvent respectivement porter les contraintes `{nouveau}` et `{détruit}`



- Les objets (et les liens) créés et détruits au cours de la même interaction portent la contrainte `{transitoire}`

Diagramme de collaboration

- Possibilité d'exprimer l'envoi répétitif de messages (éventuellement en parallèle) sur une collection d'objets



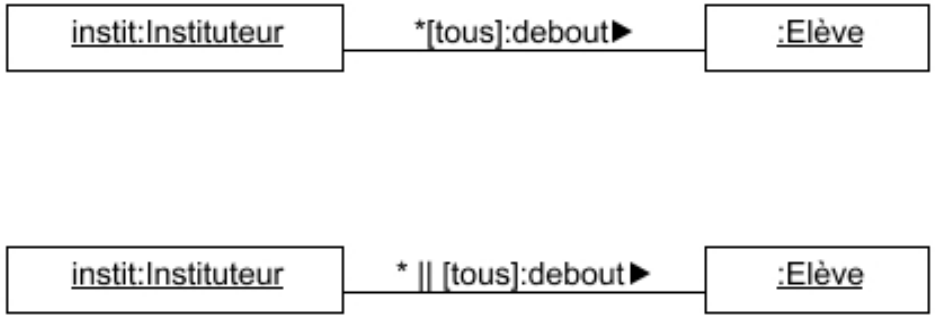


Diagramme de collaboration

- Il est possible de faire intervenir un acteur (cf. use cases) dans un diagramme de collaboration afin de représenter le comportement du système sous l'effet d'un stimuli externe

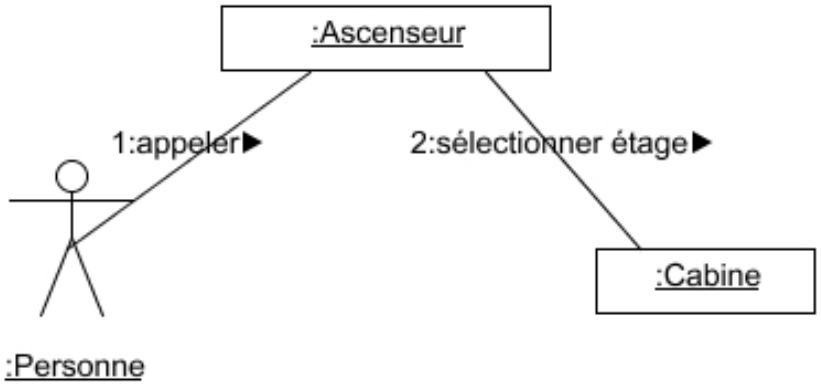


Diagramme de collaboration

- Les objets qui contrôlent le flot sont dits actifs
- Un objet actif peut activer un objet passif en lui envoyant un message ; une fois le message traité, le flot de contrôle est restitué à l'objet appelant

Ex : photocopieuse

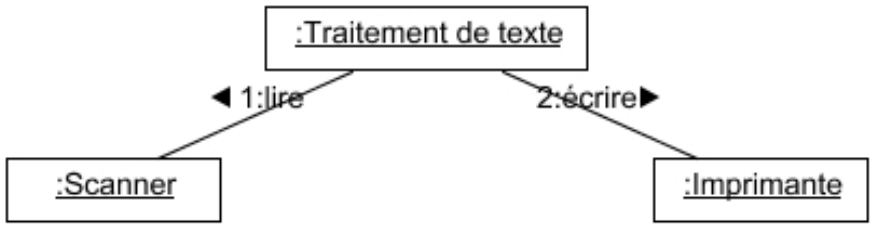


Diagramme de collaboration

- L'envoi d'un message peut être assorti d'une condition

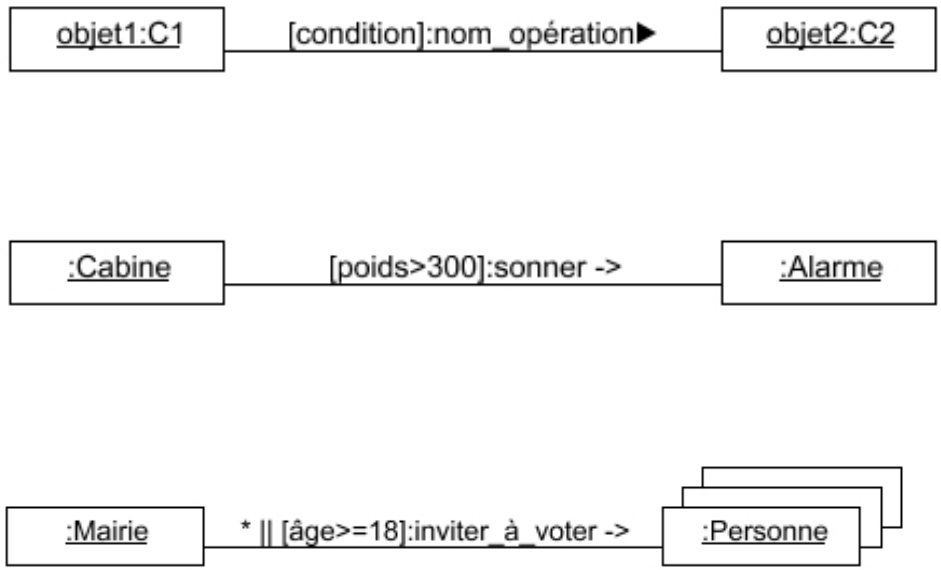
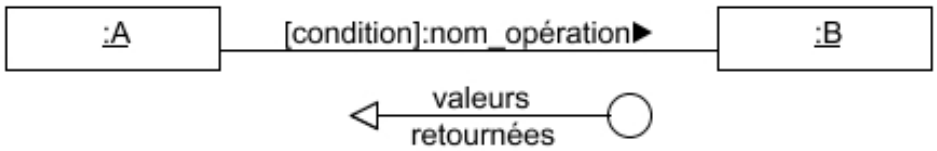


Diagramme de collaboration

- Une liste de valeurs peut être retournée suite à l'envoi d'un message



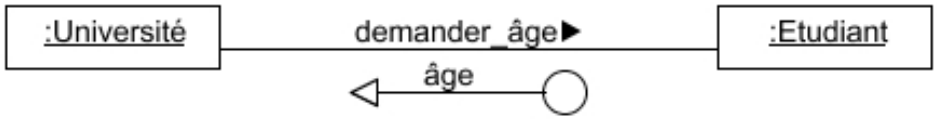


Diagramme de collaboration

Ex : diagramme de collaboration "retrait DAB"

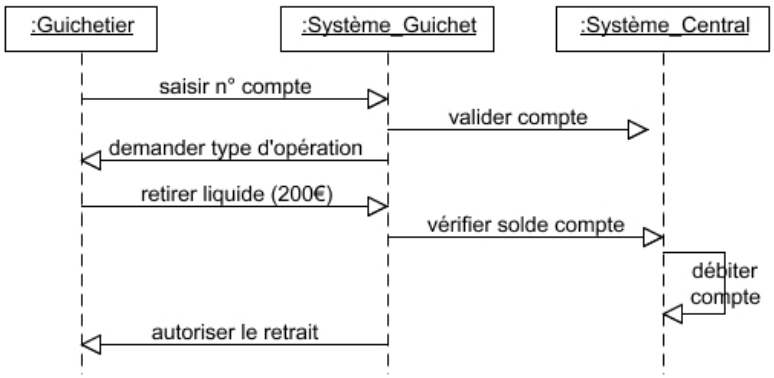


diagramme de séquence

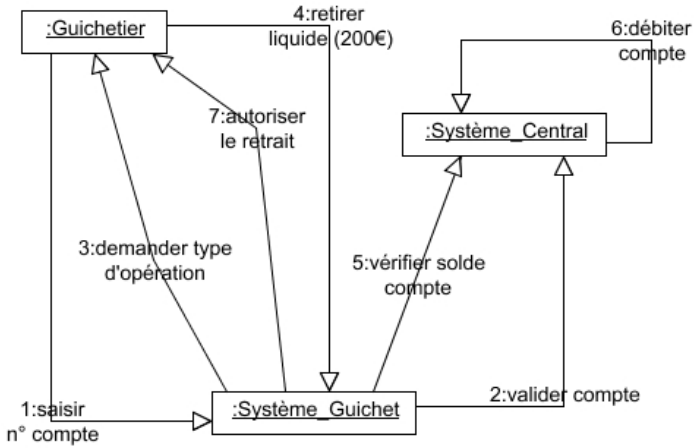
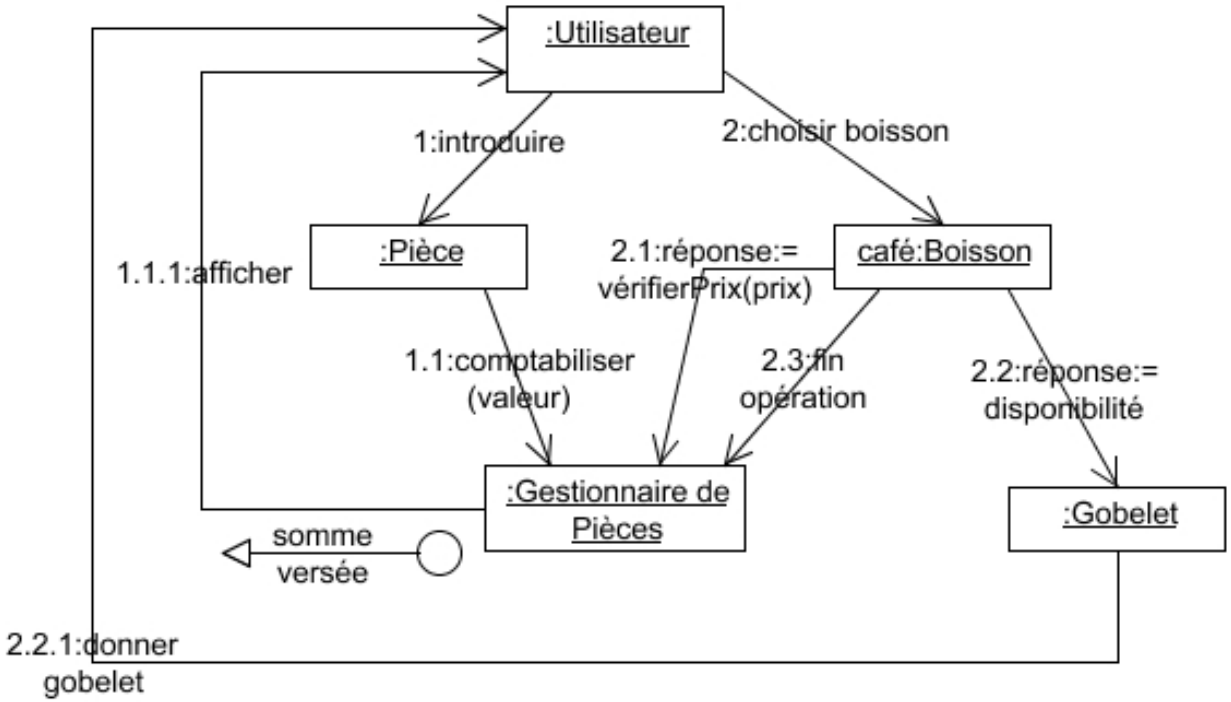


diagramme de collaboration

Diagramme de collaboration

Ex : diagramme de collaboration "demande d'une boisson disponible (café) avec introduction de la somme exacte"



Méthode MM ☺

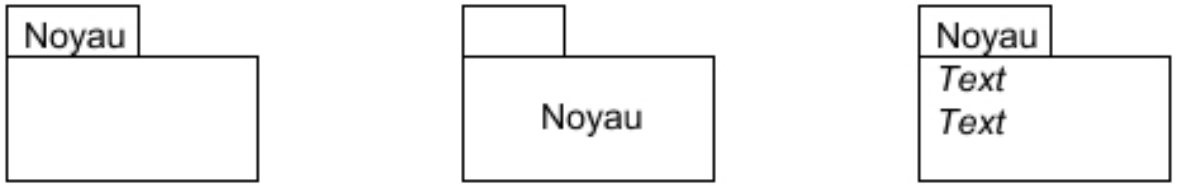
1. **Spécifier** les classes
   - cas d'utilisation
   - diagrammes de séquence
   - diagramme de classes
2. **Organiser** les composants du système
   - diagramme d'objets
   - diagrammes de collaboration
   - paquetages
   - diagrammes de composants
   - diagramme de déploiement
3. **Dynamique** des objets et des activités
   - diagrammes états-transitions
   - diagrammes d'activités

Diagramme de paquetages

- Contient
  - paquetages
  - relations
- Montre
  - l'organisation des modèles
  - les catégories du système
  - la base de l'architecture

Diagramme de paquetages

- Paquetage
  - 3 notations acceptées



- Héritage
  - au moins un des éléments du paquetage source spécialise un élément du paquetage destination

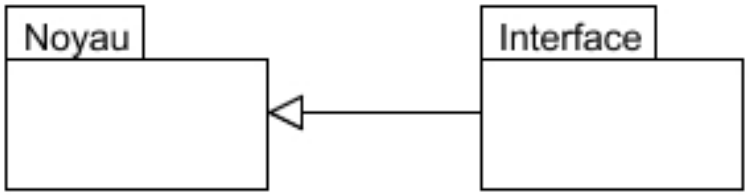
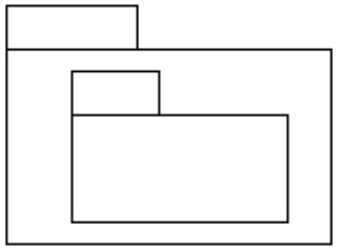


Diagramme de paquetages

- Dépendances :

- Sous paquetages :



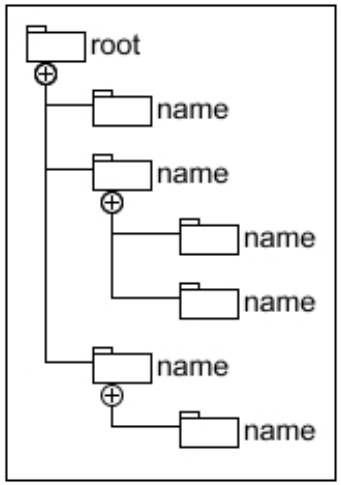


Diagramme de composants

- Contient
  - modules
- Montre
  - dépendances
  - contraintes de compilation
  - architecture physique et statique

Diagramme de composants

Ex :

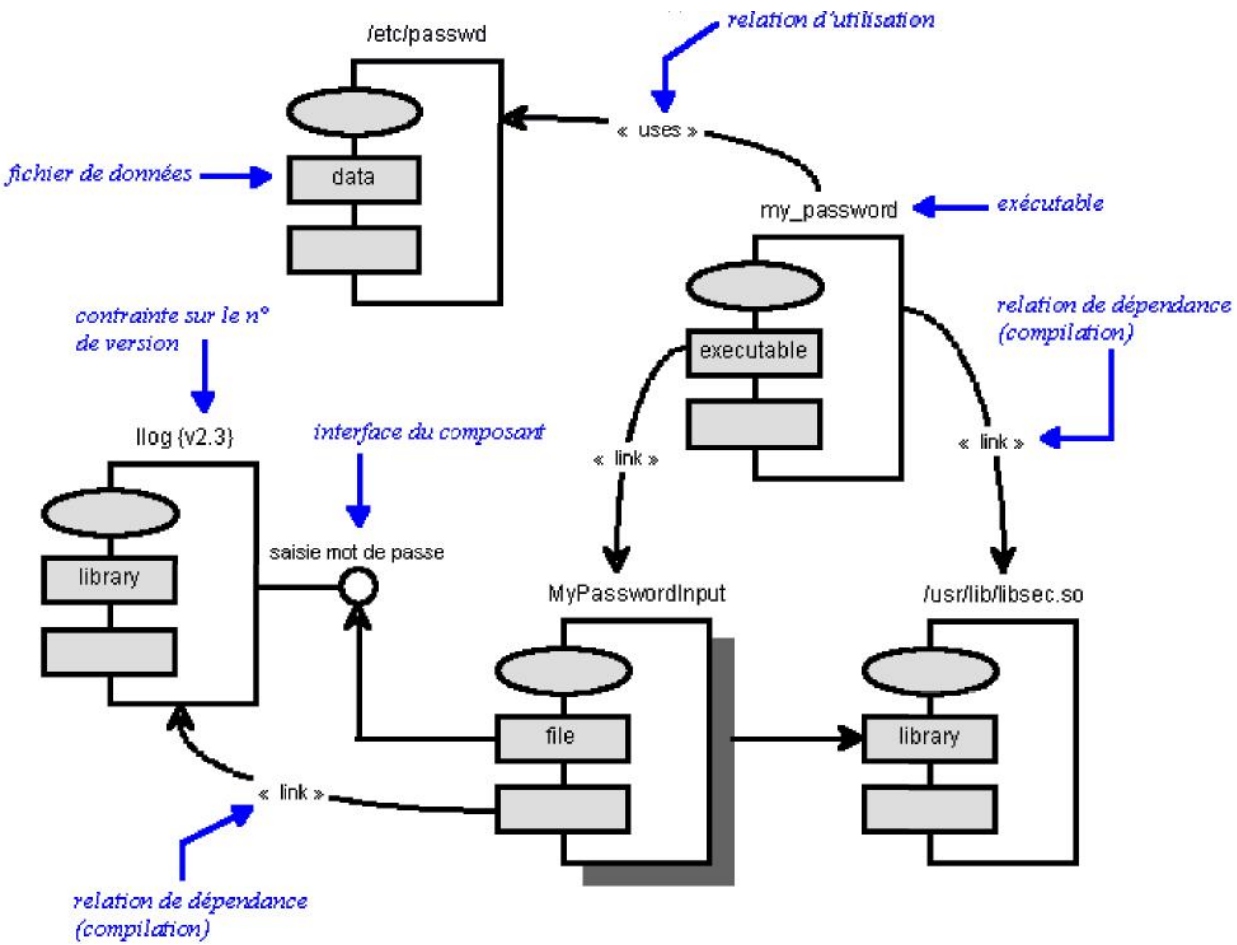
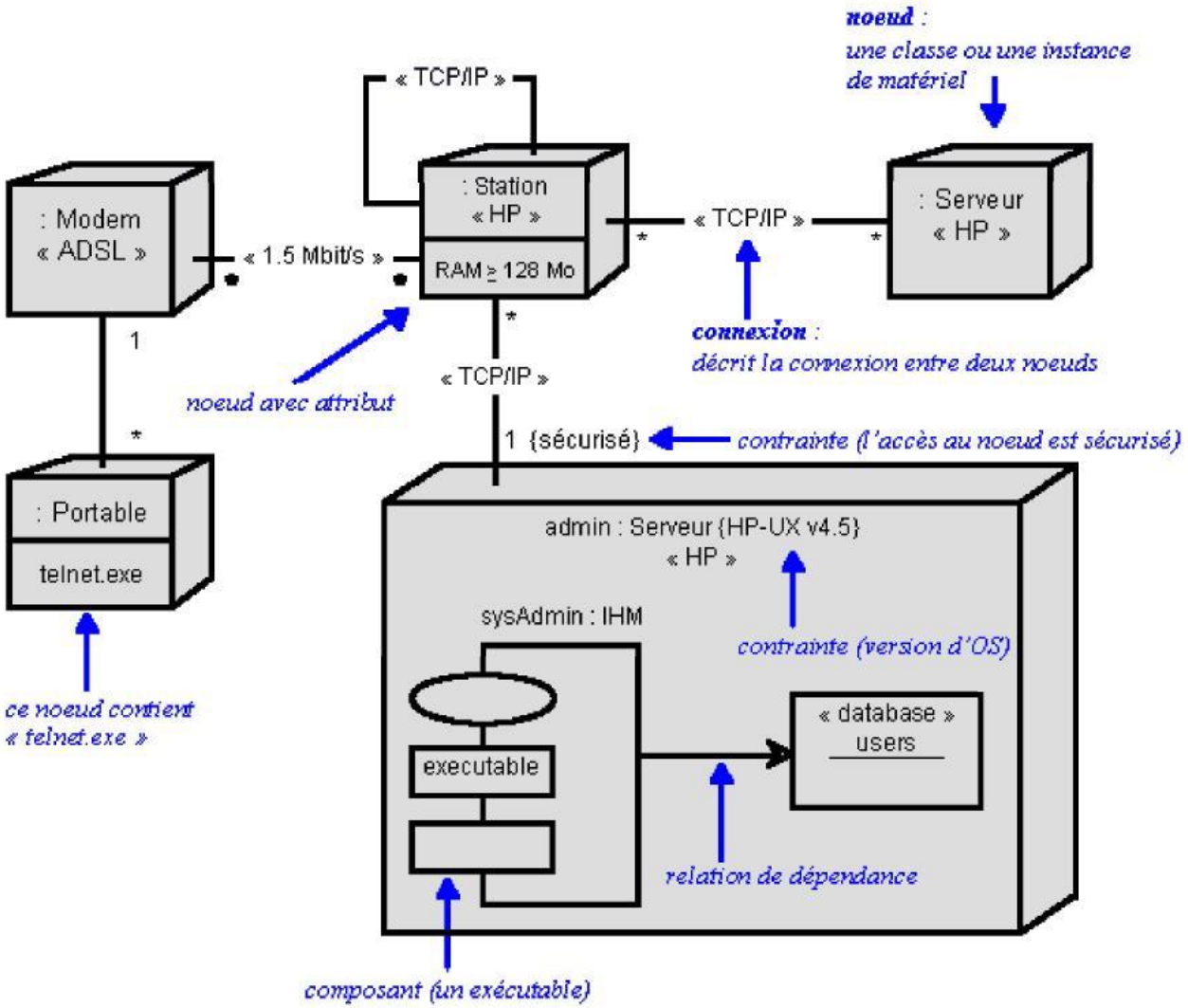


Diagramme de déploiement

- Contient
  - composants, matériels
  - connexions, nœuds
  - contraintes (débits, protocoles,. . .)
- Montre
  - disposition physique des matériels qui composent le système
  - répartition des composants sur ces matériels

Diagramme de déploiement

Ex :



Méthode MM ☺

1. **Spécifier** les classes
   - cas d'utilisation
   - diagrammes de séquence
   - diagramme de classes
2. **Organiser** les composants du système
   - diagramme d'objets
   - diagrammes de collaboration
   - paquetages
   - diagrammes de composants
   - diagramme de déploiement
3. **Dynamique** des objets et des activités
   - diagrammes états-transitions
   - diagrammes d'activités

Diagramme états-transitions

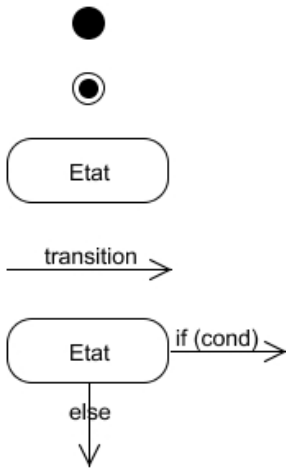
- Décrit le comportement des objets d'une classe au moyen d'un **automate d'états** associé à la classe
- Les comportement est modélisé par un graphe
  - **nœuds** = états possibles des objets
  - **arcs** = transitions d'état à état
- Une transition
  = **exécution d'une action**
  = **réaction de l'objet** (sous l'effet d'une occurrence d'un événement)

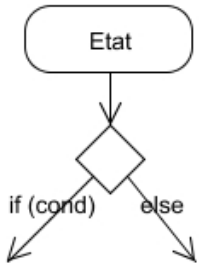
Diagramme états-transitions

- 1 état = 1 étape dans le cycle de vie d'un objet durant lequel
  - il satisfait à certaines conditions
  - il réalise certaines actions
  - ou attend certains événements
- Chaque objet possède à un instant donné son propre état
- Chaque état est identifié par un nom
- Un état est **stable** et **durable**

Diagramme états-transitions

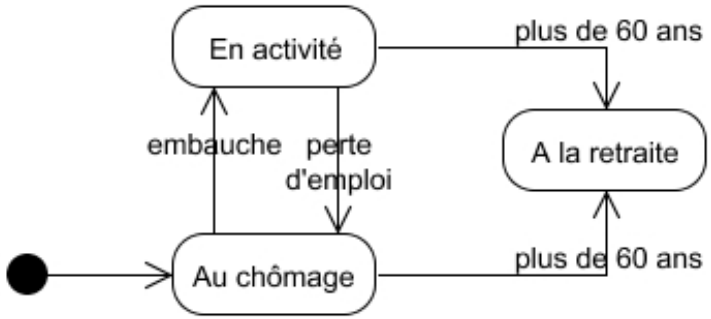
- Nomenclature
  - état initial
  - état final
  - état
  - transition
  - transition conditionnelle





autre notation

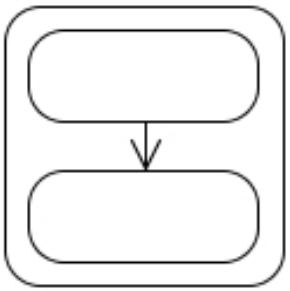
Ex : salarié



- Remarques
  - un seul et unique état initial
  - éventuellement plusieurs états finaux (voire aucun)

Diagramme états-transitions

- Remarques
  - super-état : englobe d'autres états



  - historique : mémorise le dernier sous-état actif

Ex : station de lavage

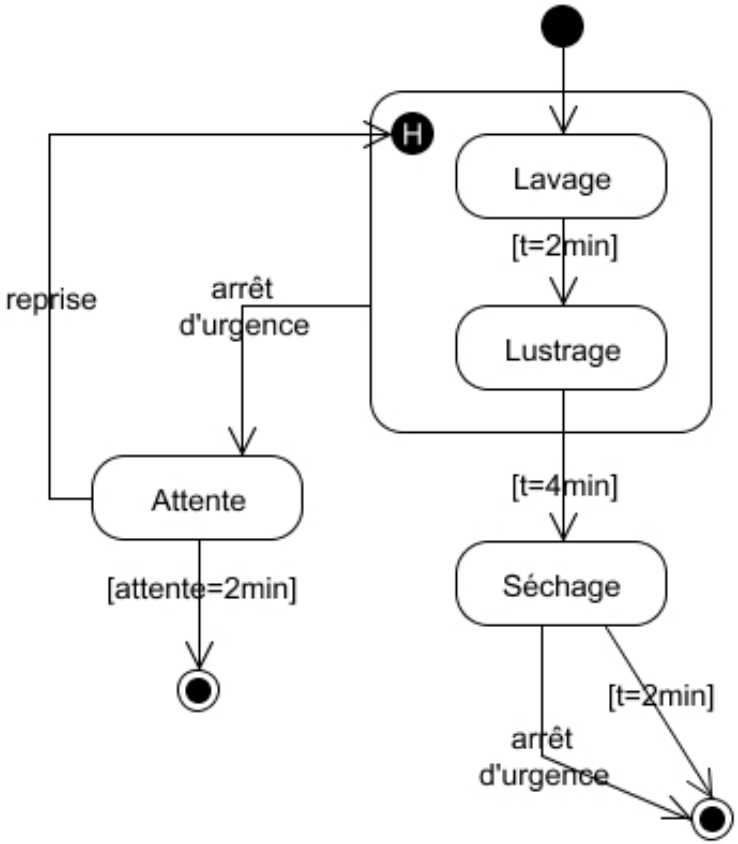
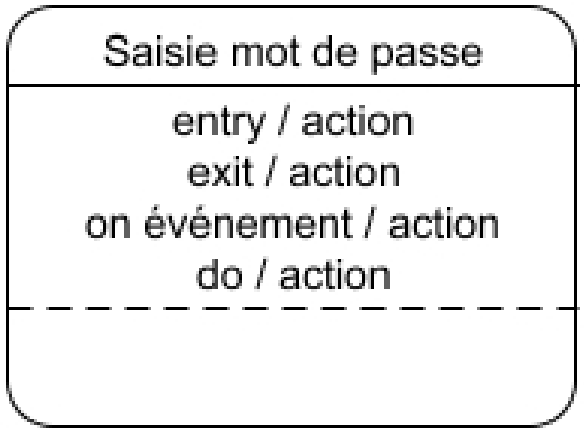


Diagramme états-transitions

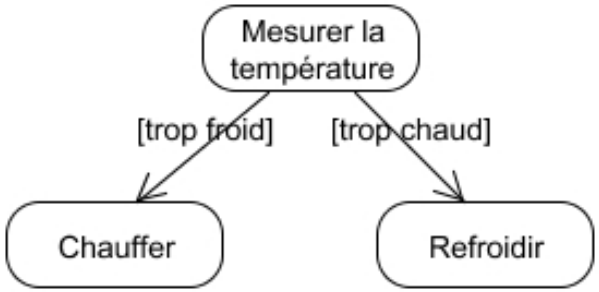
- Détailler les actions



  - `entry / action`
    ⇝ action exécutée à l'entrée de l'état
  - `exit / action`
    ⇝ action exécutée à la sortie de l'état
  - `on événement / action`
    ⇝ action exécutée à chaque fois que l'événement cité survient
  - `do / action`
    ⇝ action récurrente ou significative, exécutée dans l'état

Diagramme d'activités

- Variante des diagrammes états-transitions
  - ce diagramme met l'accent sur les activités, leurs relations et leurs impacts sur les objets
  - notion de **gardes**
    - les transitions entre activités peuvent être gardées par des conditions booléennes mutuellement exclusives
    - les gardes sont les labels des transitions dont elles valident le déclenchement



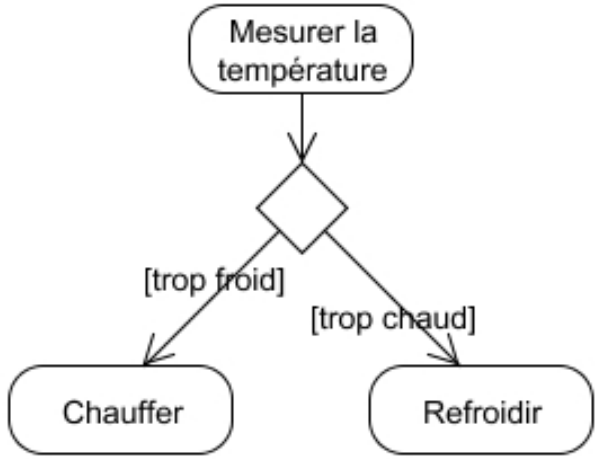
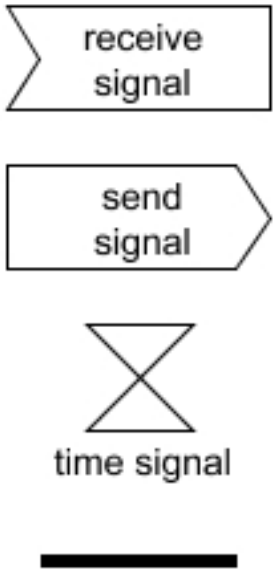


Diagramme d'activités

- Notations
  - émission d'un signal
  - réception d'un signal
  - signal temporel
  - barre de synchronisation



Ex : fin d'année...

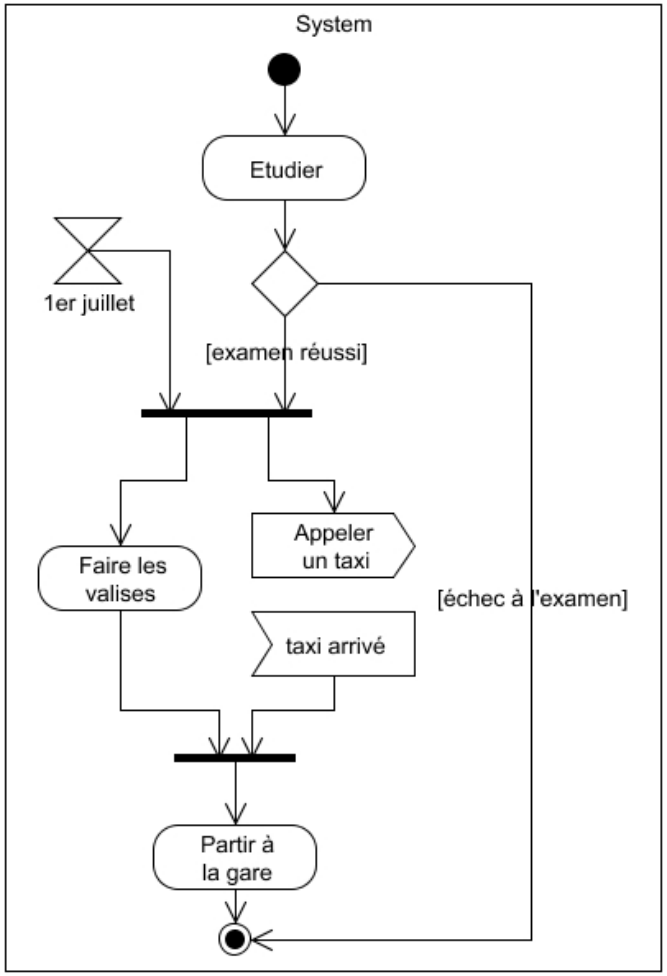


Diagramme d'activités

- Barres de synchronisation
  - → les transitions automatiques qui partent d'une barre de synchronisation ont lieu en même temps
  - → on ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent

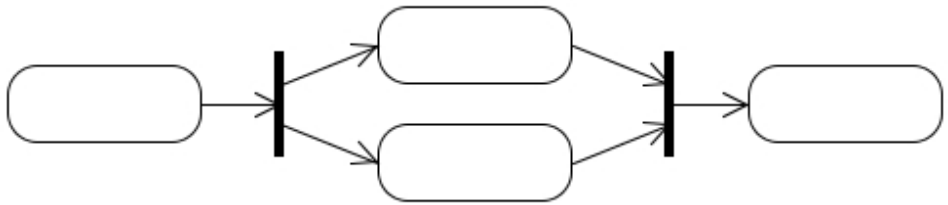
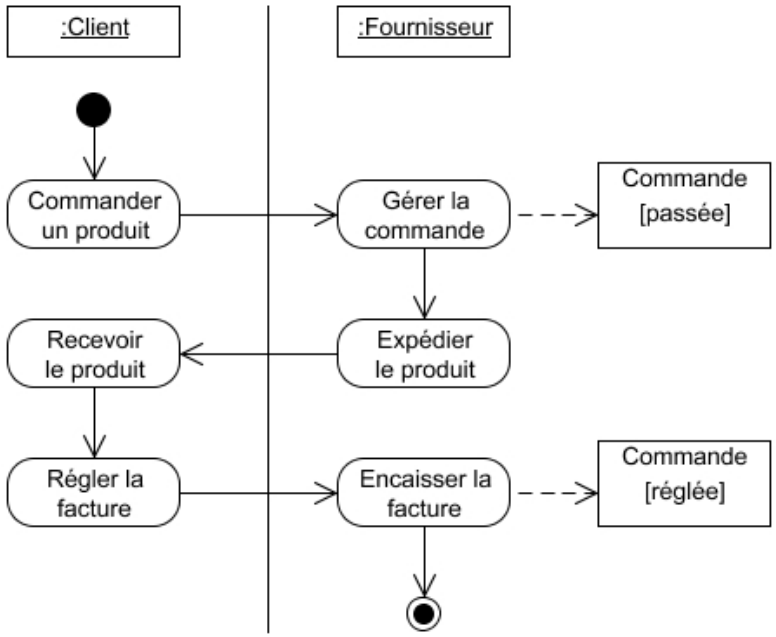


Diagramme d'activités

- Couloirs d'activités
  - les diagrammes d'activités peuvent être découpés en couloirs d'activités

  ⇒ répartition des responsabilités au sein d'un mécanisme logiciel



Méthode MM : UML ⇝ Génie Logiciel

- ✓ **Spécifier** les classes
  - cas d'utilisation
  - diagrammes de séquence
  - diagramme de classes
- ✓ **Organiser** les composants du système
  - diagramme d'objets
  - diagrammes de collaboration
  - paquetages
  - diagrammes de composants
  - diagramme de déploiement
- ✓ **Dynamique** des objets et des activités
  - diagrammes états-transitions
  - diagrammes d'activités

Plan du cours

3 Méthode de conception
- Introduction
- UML
- Compléments

Compléments

- Some free UML tools
  - UMLet (v7.1 utilisée pour ce cours)
    - http://www.umlet.com/
    - *actuellement en version 13.1 (datée du 01/01/2015)*
  - ArgoUML
    - http://fr.wikipedia.org/wiki/ArgoUML
    - http://sourceforge.net/projects/argouml.mirror/ *(v0.34 du 03/01/2012)*
  - Violet UML Editor
    - http://alexdp.free.fr/violetumleditor/page.php
    - *v2.1.0 datée du 21/01/2015 ; dispo en .deb*
  - BOUML
    - http://www.bouml.fr/
    - *v6.7.1 datée du 10/01/2015 ; dispo en dépôt pour Ubuntu*

Compléments

- Quelques liens sur UML
  - http://uml.org/
  - http://uml.free.fr/
  - http://fr.wikipedia.org/wiki/UML_(informatique)
  - http://openclassrooms.com/courses/debutez-l-analyse-logicielle-avec-uml

Passons maintenant à la pratique !