Analyse et Conception des Systèmes d'Information – Méthodes Objet

Le langage de modélisation objet UML

UML est **le** langage de modélisation de la technologie objet, standard adopté par les grands acteurs du marché.
Ce document (qui doit beaucoup aux ouvrages – que je vous conseille fortement – *De MERISE à UML* de N. Kettani, D. Mignet, P. Paré et C. Rosenthal-Sabroux et *Modélisation objet avec UML* de P.-A. Muller, tous deux parus aux éditions Eyrolles en 1998 … et à quelques recherches sur le Web) propose une présentation rapide d'UML (qui renvoie surtout à des ouvrages et à des sites Internet) et la description des modèles d'UML (illustrés de surcroît par l'exemple « jouet »).

1. Présentation

UML, langage de modélisation objet, est récent mais déjà très référencé (qu'il s'agisse d'ouvrages ou de sites Internet) et dispose de nombreux outils. Notez qu'UML est ouvert et n'est la propriété de personne.
Après avoir cité quelques méthodes objet, ce chapitre présente succinctement UML : une définition, des généralités, un court historique, une bibliographie et des outils (i.e. ateliers de génie logiciel).

1.1. Quelques méthodes objet

Pas moins d'une cinquantaine de méthodes objet ont été dénombrées au milieu des années 90.

> Les méthodes objet qui suivent sont citées par J. Delatour (LIGLOO) du LAAS de Toulouse (http://www.laas.fr/~delatour/Igloo/methodeOO_image_fr.html) ou documentées dans N. Kettani, D. Mignet, P. Paré et C. Rosenthal-Sabroux, *De MERISE à UML*, Eyrolles, 1998.

Catalysis, **D. D'Souza**.

- D. D'Souza et A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998, http://iconcomp.com/catalysis/catalysis-book/index.html.
- http://www.iconcomp.com/catalysis/

HOOD (*Hierarchical Object Oriented Design*), **B. Delatte, M. Heitz et J.F. Muller**, 1987.

- B. Delatte, M. Heitz et J.F. Muller, *HOOD Reference manual 3.1*, Masson, 1993.
- J.-P. Rosen, *HOOD, An Industrial Approach for Software Design*, édition HOOD User Group, http://perso.wanadoo.fr/adalog/hoodbook.htm.
- http://www.hood.be : site officiel de HOOD
- http://www.estec.esa.nl/wmwww/WME/oot/hood/ : site de l'ESA (Agence Spatiale Européenne)

M2PO.

- D. R. C. Hill, *Analyse orientée objets & modélisation par simulation*, Addison-Wesley France, 1993.

OMT (*Object Modeling Technique*), **J. Rumbaugh**, 1987-1989.

- J. Rumbaugh, M. Blaha, W. Premerlani et F. Eddy, *OMT Modélisation et Conception Orientées Objet*, Masson, 1996 (2nde éd.).
- J. Rumbaugh et al., *OMT Solution des exercices*, Masson, 1996.
- http://www.csioo.com/cetusfr/oo_ooa_ood_methods.html#oo_meth_omt : Cetus
- http://www.rational.com/support/techpapers/omt/ : Rational Software Corporation
- http://www.rational.com/support/techpapers/omt/summary.html : J. Rumbaugh
- http://www.netinfo.fr/objectland/langages/n9.94/OMT.html : en français (partie statique)

OOA (*Object Oriented Analysis*), **S. Shlaer & S. J. Mellors**, 1979.

- S. Shlaer et S. J. Mellors, *Object lifecycles: Modeling the world in states*, Yourdon Press, Prentice Hall, 1988.

OOA/OOD (*Object Oriented Analysis/Object Oriented Design*), **P. Coad & E. Yourdon**, 1991.

- P. Coad et E. Yourdon, *Object Oriented Analysis*, Yourdon Press, Prentice Hall, 1991 (2nde éd.).

OOD (*Object Oriented Design*), **G. Booch**, entre 1980 et 1983.

- G. Booch, *Analyse et Conception Orientées Objet*, Addison-Wesley France, 1994 (2nde éd.).
- R. Martin, *Designing Object Oriented C++ Applications using the Booch Method*, Prentice Hall, 1995, http://www.oma.com/DOOCPPAUBM/bookFlier.html.
- http://www.csioo.com/cetusfr/oo_ooa_ood_methods.html#oo_meth_booch : Cetus
- http://www.iconcomp.com/papers/comp/comp_87.html : ICON Computing
- http://www.platinum.com/clrlake/para_30/oo_mthd/booch.htm : Platinium Technology
- http://www.itr.ch/courses/case/BoochReference/ : (par P. Schneider)

- http://arkhp1.kek.jp/~amako/amakoInfo.html : (par K. Amako)

OOM (*Orientaton Objet dans Merise*), **A. Rochfeld et M. Bouzeghoub**, 1993.

- A. Rochfeld et M. Bouzeghoub, *From Merise to OOM*, revue ISI vol. 1 n° 2, AFCET-Hermès, Paris, 1993.

OOSE (*Object Oriented Software Engineering*), **I. Jacobson**, 1980.

- I. Jacobson, M. Christerson, P. Jonson et G. Ôvergaard, *Object Oriented Software Engineering: A use case driven approach*, Addison-Wesley, 1992.

OPEN.

Concurrent d'UML, proposant en plus un processus de développement.

- D. Firesmith, B. Henderson-Sellers et I. Graham, *The OML Reference Manual*, SIGS Books, NY, 1997 (ou chez Cambridge University Press).
- D.G. Firesmith, G. Hendley, S. Krutsch et M. Stowe, *Documenting a Complete Java Application using OPEN*, série OPEN publiée par Addison-Wesley.
- I. Graham, B. Henderson-Sellers et H. Younessi, *The OPEN Process Specification*, Addison-Wesley,1997.
- B. Henderson-Sellers,T. Simons et H. Younessi, *The OPEN Toolbox of Techniques*, Masson, Addison-Wesley, 1998.
- http://www.csse.swin.edu.au/cotar/OPEN/
- http://www.csse.swin.edu.au/cotar/OPEN/OBJMAG/OBJMAG.html
- http://www.csse.swin.edu.au/cotar/OPEN/ROAD/ROAD.html

ROOM.

- http://www.dmi.usherb.ca/~normand/memoire/index.html : (par M. Normandeau)

Et quelques autres méthodes objet : **Fusion**, **JSD** (*Jackson System Development*) de M. Jackson, **MACH_2** (*Machine Abstraite de Conception Hiérarchique orientée objet*), **Objectory**, **Octopus** et **SYS_P_O** (*Systems Project Object*) de P. Jaulent.

1.2. Un langage unifié pour la modélisation objet

UML (*Unified Modeling Language*) est un langage unifié pour la modélisation objet.

- UML est un **langage** (**de modélisation** objet) et propose donc une notation et une sémantique associée à cette notation (i.e. des modèles), mais pas de processus (i.e. de démarche proposant un enchaînement d'étapes et d'activités qui mènent à la résolution d'un problème posé) ; UML n'est donc pas une méthode.
- UML **unifie** des méthodes objet, et plus particulièrement les méthodes Booch'93 de G. Booch, OMT-2 (Object Modeling Technique) de J. Rumbaugh et OOSE (Object-Oriented Software Engineering) d'I. Jacobson. Actuellement, ces trois personnes (surnommées les « trois amigos ») travaillent pour Rational Software Corporation.
  UML reprend en particulier les notions de partitions en sous-systèmes de Booch'93, de classes et d'associations d'OMT-2, et d'expression des besoins par les interactions entre les utilisateurs et le système d'OOSE.
- UML a une approche entièrement (i.e. couvrant tout le cycle de développement : analyse, conception et réalisation) **objet** (et non fonctionnelle) : le système est décomposé en objets collaborant (plutôt qu'en tâches décomposées en fonctions plus simples à réaliser).

1.3. Quelques généralités

UML est conçu pour modéliser divers types de systèmes, de taille quelconque et pour tous les domaines d'application (gestion, scientifique, temps réel, système embarqué).
UML permet de diviser le système d'information (d'une organisation) en le système métier et le système informatique. Le système métier modélise les aspects statiques et dynamiques de l'activité selon une vision externe et une vision interne (en ignorant l'implémentation technique) tandis que le système informatique recouvre la partie automatisée du système métier concrétisant les choix effectués parmi les différentes technologies d'actualité. Les concepts manipulés sont les mêmes, à chacun de ces deux niveaux d'abstraction.
UML est fortement inspiré de l'approche 4+1 vues (logique, composants, processus, déploiement et cas d'utilisation) indépendantes définies par P. Kruchten pour exprimer les diverses perspectives de l'architecture d'un système informatique.
UML se compose d'une part des éléments de modélisation qui représentent toutes les propriétés du langage et d'autre part des diagrammes (de cas d'utilisation, de classes, d'objets, d'états-transitions, d'activités, de séquence, de collaboration, de composants et de déploiement) qui en constituent l'expression visuelle et graphique.
UML n'impose pas de processus de développement logiciel particulier, même si celui sous-jacent est un processus itératif (précisant à chaque itération les degrés d'abstraction), incrémental (i.e. en divisant le développement en étapes aboutissant chacune à la construction de tout ou partie du système), centré sur l'architecture (au niveau de la modélisation comme de la production), conduit par les cas d'utilisation (modélisant l'application à partir des modes d'utilisation attendus par les utilisateurs), piloté par les risques (afin d'écarter les causes majeures d'échec) tel que le 2TUP (*Two Tracks Unified Process*) présenté notamment dans l'ouvrage *UML en action – De l'analyse des besoins à la conception en Java* de P. Roques et F. Vallée paru aux éditions Eyrolles en 2000.
UML prend en compte de manière complètement intégrée l'ingénierie des besoins (cas d'utilisation).
UML est automatisable pour générer du code à partir des modèles vers les langages et les environnements de programmation.
UML est générique, extensible (en plus de couvrir les possibilités des différentes technologies objet existantes) et configurable.
UML se veut intuitif, simple et cohérent.

1.4. Historique

UML est né en octobre 1994 chez Rational Software Corporation à l'initiative de G. Booch et de J. Rumbaugh.
UML 1.1 a été **standard**isé par l'OMG (Object Management Group) le 17 novembre 1997 suite à la demande émanant de la collaboration de plusieurs entreprises (Hewlett-Packard, IBM, i-Logix, ICON Computing, IntelliCorp, MCI Systemhouse, Microsoft, ObjecTime, Oracle, Platinum Technology, Ptech, Rational Software Corporation, Reich Technologies, Softeam, Sterling Software, Taskon et Unisys).
La version actuelle (depuis juin 1999) est UML **1.3** (la version 1.4 sera bientôt prête, afin de préparer la prochaine version 2.0).

1.5. Bibliographie

La littérature est très abondante (en langue française, et davantage encore en langue anglaise). Consultez aussi http://www.uml.db.informatik.uni-bremen.de/umlbib/.
La **documentation** (808 pages au format `pdf`, en langue anglaise) est entièrement disponible sur le site www.rational.com (`uml13.zip`, 2.7 Mo) de Rational Software Corporation. Tous les concepts d'UML sont eux-mêmes modélisés en UML (méta-modèle).
En plus de la documentation accessible sur Internet, d'autres sites et groupes de discussion (*news*) vous donneront de nombreuses informations.

1.5.1. Ouvrages en langue française

- J. Gabay, *Merise. Vers OMT et UML*, InterÉditions, 1998.
- N. Kettani, D. Mignet, P. Paré et C. Rosenthal-Sabroux, *De MERISE à UML*, Eyrolles, 1998 : **recommandé** !
- M. Lai, *Penser objet avec UML et Java*, InterÉditions, 1998.
- M. Lai, *UML : La notation unifiée de modélisation objet – De Java aux EJB*, Dunod, 2000
- N. Lopez, J. Migueis et E. Pichon, *Intégrer UML dans vos projets*, Eyrolles, 1997.
- C. Morley, B. Leblanc et J. Hugues, *UML pour l'analyse d'un système d'information – Le cahier des charges du maître d'ouvrage*, Dunod, 2000.
- P.-A. Muller, *Modélisation objet avec UML*, Eyrolles, 1998 : souvent cité et **conseillé** !
- P. Roques et F. Vallée, *UML en action – De l'analyse des besoins à la conception en Java*, Eyrolles, 2000.
- C. Soutou, *Objet-Relationnel sous Oracle8, Modélisation avec UML*, Eyrolles, 1999.

1.5.2. Les références (ouvrages et articles cités dans la documentation d'UML 1.3)

- C. Bock et J. Odell, "A Foundation For Composition", *Journal of Object-Oriented Programming*, oct. 1994.
- G. Booch, J. Rumbaugh et I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- S. Cook et J. Daniels, *Designing Object Systems: Object-oriented Modelling with Syntropy*, Prentice Hall Object-Oriented Series, 1994.
- D. D'Souza et A. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.
- M. Fowler avec K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
- M. Griss, "Domain Engineering And Variability In The Reuse-Driven Software Engineering Business", *Object Magazine*, déc. 1996.
- D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, 1987, pp. 231-274.
- D. Harel et E. Gery, "Executable Object Modeling with Statecharts", *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, mars 1996, pp. 246-257.
- D. Harel et A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Trans. Soft. Eng. Method 5*:4, oct. 1996.
- I. Jacobson, G. Booch et J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
- R. Malan, D. Coleman, R. Letsinger et al., "The Next Generation of Fusion", *Fusion Newsletter*, oct. 1996.
- J. Martin et J. Odell, *Object-Oriented Methods, A Foundation*, Prentice Hall, 1995.
- G. Ramackers et D. Clegg, "Object Business Modelling, requirements and approach" in Sutherland, J. and Patel, D. (eds.), *Proceedings of the OOPSLA95 Workshop on Business Object Design and Implementation*, Springer Verlag.
- G. Ramackers et D. Clegg, "Extended Use Cases and Business Objects for BPR", ObjectWorld UK '96, Londres, 18-21 juin 1996.
- J. Rumbaugh, I. Jacobson et G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
- B. Selic, G. Gullekson et P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.
- J. Warmer et A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

1.5.3. Sites Internet

Quelques sites de référence.
- www.omg.org : site de l'**OMG**
- http://uml.shl.com : site de l'OMG UML *Revision Task Force*
- http://www.rational.com/uml : site de **Rational** Software Corporation

- http://uml.free.fr/ : **cours en français** (de L. Piechocki)

Certaines des entreprises ayant participé au projet.

- www.ilogix.com : I-Logix (dont D. Harel)
- www.rational.com : Rational Software Corporation (dont G. Booch, J. Rumbaugh et I. Jacobson)
- www.softeam.fr : Softeam (dont P. Desfray)
- www.valtech.com : Valtech

Sites universitaires (français).

- http://www.enib.fr/~chevaill/Course/ModuleGL/UML/index.html : ENIB (École Nationale d'Ingénieurs de Brest) (par P. Chevaillier)
- http://www.essaim.univ-mulhouse.fr/uml : ESSAIM (École Supérieure des Sciences Appliquées pour l'Ingénieur - Mulhouse)
- http://www.iutc3.unicaen.fr/~moranb/cours/acsi/menucoo.htm : IUT de Caen (par B. Morand)
- http://www.laas.fr/~delatour/Igloo/index_image_fr.html : LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes) de Toulouse, LIGLOO (Liens sur le Génie Logiciel Orienté Objet) (par J. Delatour)
- http://lrim.sciences.univ-metz.fr/~lanuel/UML/Cours/sld001.htm : université de Metz (par Y. Lanuel)
- http://www-igm.univ-mlv.fr/~dr/DESS/glossaireGraphique/GlossaireGraphique.htm : université de Marne-la-Vallée (par D. Revuz)
- http://www.unantes.univ-nantes.fr/~cerisier/uml/www.html : université de Nantes, IAE, Laboratoire de Recherche en Sciences de Gestion (par F. Cerisier avec J. Bézivin)
- http://www.univ-pau.fr/~bruel/Recherche/SIC/sic99.html : université de Pau et des Pays de l'Adour (par J.-M. Bruel)
- http://perso.wanadoo.fr/alsoftware/french/formation/uml/uml.html : Lycée Diderot (2ème année du BTS Informatique Industrielle) de Paris

Des sites … en vrac (en français) !

- http://www.caminao.com/documentation.htm : le projet Caminao a pour ambition de créer une communauté pédagogique autour d'UML.
- http://medit.epfl.ch:4444/visitors/events/meetings/11fevrier/epfl2/index.htm : UML - application au projet
- http://www.esil.univ-mrs.fr/~salenson/projets/juml.htm : extension d'UML pour JAVA : J-UML
- http://www-edu.gel.usherb.ca/nkoj01/gei450/projet/Uml/survol.html : survol de la notation unifiée UML

D'autres sites en vrac.

- http://www.isg.de/people/marc/UmlDocCollection/UMLDocCollection.html
- http://jeffsutherland.org/
- http://www.krumbach.de/home/jeckle/unified.htm
- http://www.objectnews.com/
- http://iamwww.unibe.ch/~scg/OOinfo/

Et quelques sites sur l'objet.

- http://www.cetus-links.org/ : (The CETUS links)
- http://www.stm.tj/objet/ : (La boîte à objets)

1.5.4. Groupes de discussion

- `comp.objet`
- `comp.software-eng`
- `fr.comp.objet`

1.6. Outils

> Les outils (AGL) qui suivent sont cités par P. Roques de Valtech Toulouse (http://www.essaim.univ-mulhouse.fr/uml/outillage/outillage.html).

Quelques outils.

- `Argo/UML` (http://argouml.tigris.org/index.html) de l'université de Californie UCI (www.ics.uci.edu/pub/arch/uml)
- `Prosa/om` (www.prosa.fi/prosa.html) d'Insoft Oy
- `Objecteering` (http://www.objecteering.com/) de Softeam (www.softeam.fr)
- `Paradigm Plus` (www.platinum.com/products/appdev/pplus_ps.htm) de Computer Associates
- `Rhapsody` (www.ilogix.com/fs_prod.htm) d'I-Logix (www.ilogix.com)
- `Rose 2000` (http://www.rosearchitect.com/) de Rational Software Corporation (www.rational.com)
- `StP/UML` (www.aonix.com/Products/SMS/core7.1.html) d'Aonix (www.aonix.fr)
- `Visual UML` (www.visualuml.com/products.htm) de Visual Object Modelers

D'autres AGL.

- `4Keeps` d'A. D. Experts (www.adexperts.com)
- `Bold for Delphi` (www.boldsoft.com/products) de BoldSoft
- `COOL:Jex` (www.cool.sterling.com/products/Jex/index.htm) de Sterling Software
- `Elixir CASE` (www.elixirtech.com/ElixirCASE/index.html) d'Elixir Tech.

- `Ensemble Suite` (www.ensemble-systems.com/products.html) d'Ensemble Systems
- `Florist` (www.qoses.com/products) de Qoses
- `FrameWork` de Ptech (www.ptechinc.com)
- `Framework Studio` (www.blueprint-technologies.com/products/index.html) de Blueprint Tech.
- `GDPro` d'Advanced Software Tech. (www.advancedsw.com)
- `How` de Riverton Software (www.riverton.com)
- `Innovator` (www.mid.de/e/innovato/index.htm) de Mid
- `ISOA` de Mega International (www.mega.com)
- `JVision` d'Object Insight (www.objectinsight.com)
- `MagicDraw UML` de No Magic (www.nomagic.com)
- `Mesa/Vista Enterprise` (www.mesasys.com/vista/vista_family.html) de Mesa
- `MetaEdit+` de Metacase (www.metacase.com)
- `Model Prototyper` d'ObjeXion (www.objexion.com)
- `Object Domain` d'Object Domain Systems (www.objectdomain.com)
- `ObjectGeode` (www.verilogusa.com/products/geode.htm) de Verilog
- `ObjectiF` (www.microtool.de/obje.e) de MicroTOOL
- `Opentool` de TNI (www.tni.fr)
- `ORCA` (www.telelogic.com/solution/tools/orca.asp) de Telelogic
- `Pragmatica` de Pragmatix Software (www.pragsoft.com)
- `Real-Time Studio Prof.` d'Artisan (www.artisansw.com)
- `Rocase` (www.cs.ubbcluj.ro/rocase) de l'université de Roumanie Babes-Bolyai
- `Rose RealTime` de Rational Software Corporation (www.rational.com)
- `RoZeLink` d'Headway Software (www.headway-software.com)
- `Select/Enterprise` (www.princetonsoftech.com/products) de Princeton Softech
- `Simply Objects` (www.adaptive-arts.com/products.htm) d'Adaptive
- `SoftModeler/Business` (www.softera.com/products.htm) de Softera
- `Structure Builder` de Tendril Software (www.tendril.com/)
- `System Architect` (www.popkin.com/products/sa2001/product.htm) de Popkin
- `Together/Enterprise` (www.togethersoft.com) d'Object International
- `UML Designer` (www.software.ibm.com/ad/smalltalk/about/umldfact.html) d'IBM
- `Visual Modeler` (msdn.microsoft.com/vstudio/prodinfo/new.asp) de Microsoft
- `Visual Thought` de Confluent (www.confluent.com)
- `Win A&D` d'Excel Software (www.excelsoftware.com)
- `WithClass` de Microgold Software (www.microgold.com)
- `Xtend:Specs` (www.iconcomp.com/products/index.html) d'ICON Computing

Et des sites pour en trouver davantage encore.

- http://www.krumbach.de/home/jeckle/unified.htm : page de M. Jeckle
- http://www.laas.fr/~delatour/Igloo/index_image_fr.html : page LIGLOO
- http://www.stm.tj/objet/ : (La boîte à objets)

2. Notions communes aux différents modèles

Ce chapitre présente toutes les notions communes aux modèles, à savoir des définitions générales (cycle de vie, etc.) et les éléments de modélisation (nom, étiquette, expression, contrainte, propriété, commentaire, note, paquetage, sous-système, modèle, stéréotype, etc.).

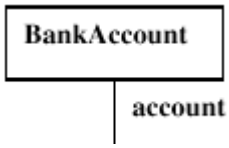
2.1. Définitions générales

- domaine (*domain*) : champ d'application
- élément de modélisation (*model element*) : représentation d'une abstraction issue du domaine du problème
- objet (*object*) : entité ayant une frontière et une identité bien définies qui encapsule un état et un comportement
- classe (*class*) : description d'un ensemble d'objets qui partagent les mêmes caractéristiques
- système (*system*) : ensemble d'objets connectés et organisés pour atteindre un but spécifique
- modèle (*model*) : abstraction sémantique fermée d'un système
- cycle de vie (*life cycle*) : étapes du développement et ordonnancement de ces étapes
- analyse (*analysis*) : détermination du « quoi » et du « quoi faire » d'une application
- conception (*conception*) : détermination du « comment » d'une application

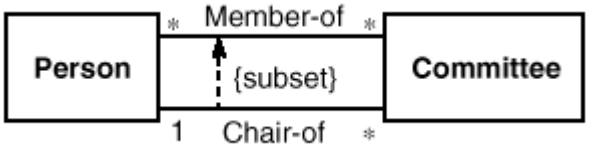
- modélisation (*modeling*) : synonyme d'analyse, et par extension, élaboration des modèles (y compris en conception)
- méta-modèle (*metamodel*) : modèle qui décrit ses éléments de modélisation
- méta-modélisation (*metamodeling*) : modélisation récursive des éléments de modélisation à partir d'eux-mêmes
- spécification (*specification*) : description exhaustive d'un élément de modélisation
- interface (*interface*) : partie visible d'une classe, d'un groupe d'objets (parfois utilisé comme synonyme de spécification)
- documentation (*documentation*) : représentation textuelle des modèles
- diagramme (*diagram*) : représentation graphique d'éléments de modélisation
- notation (*notation*) : ensemble des signes et symboles qui composent le langage UML (i.e. ensemble des représentations graphiques et textuelles qui constituent les diagrammes)
- revue : révision formelle d'une documentation, d'un modèle
- test (*test*) : ensemble des mesures et des activités qui visent à garantir le fonctionnement satisfaisant du logiciel
- couche : segmentation horizontale des modèles
- vue (*view*) : projection des éléments de modélisation d'un (ou plusieurs) modèle(s) selon une certaine perspective

2.2. Éléments de modélisation

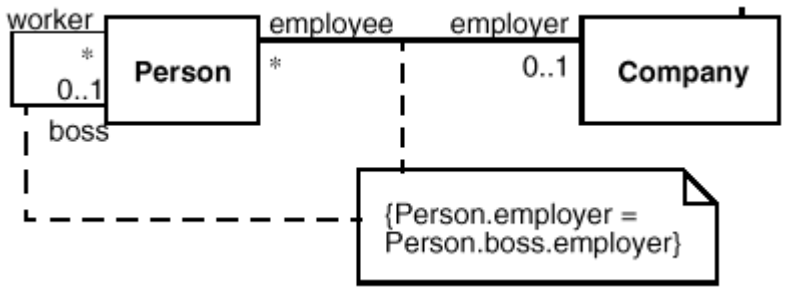
- chaîne (*string*) : suite de caractères
  Exemples de chaînes :
  ```
  BankAccount
  integrate (f: Function, from: Real, to: Real)
  { author = "Joe Smith", deadline = 31-March-1997, status = analysis }
  ```
- nom (*name*) : chaîne utilisée pour identifier un élément d'un modèle
  Exemples de noms :
  ```
  BankAccount
  integrate
  controller
  abstract
  this_is_a_very_long_name_with_underscores
  ```
  Exemple de chemin d'accès (*pathname*) :
  ```
  MathPak::Matrices::BandedMatrix
  ```
- étiquette (*label*) : chaîne attachée à un symbole graphique
  Exemple d'étiquette : l'étiquette `BankAccount` est attachée en étant contenue dans la boîte (*containment*) tandis que l'étiquette `account` est attachée par adjacence (*adjacency*).



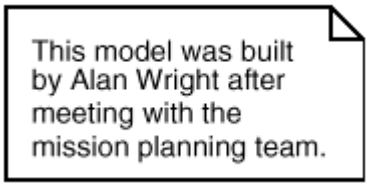
- mot-clé (*keyword*)
  Format d'un mot-clé : « ◁`mot-clé`▷ »
- expression (*expression*) : chaîne qui évalue une valeur (d'un type particulier)
  Exemples d'expressions :
  ```
  BankAccount
  BankAccount * (*) (Person*, int)
  array [1..20] of reference to range (-1.0..1.0) of Real
  [ i > j and self.size > i ]
  ```
  Exemples du langage OCL (*Object Constraint Language*) :
  ```
  flight.pilot.training_hours > flight.plane.minimum_hours
  company.employees->select (title = "Manager" and self.reports->size > 10)
  ```
- collection (*collection*) : regroupement d'objets (terme générique qui ne précise pas la nature du regroupement)
- cardinalité (*cardinality*) : nombre d'éléments dans une collection
- multiplicité (*multiplicity*) : spécification d'échelle qui précise les cardinalités autorisées
- type primitif (*primitive type*) : type prédéfini (booléen, expression, liste, chaîne, point, nom, multiplicité, temps, non interprété).
- contrainte (*constraint*) : relation sémantique sur des éléments de modélisation qui spécifie une proposition devant être vérifiée
  Format d'une contrainte : `{` ◁`contrainte`▷ `}`
  1er exemple de contrainte : le président de chaque comité doit être l'un des membres de ce comité (`{subset}`).



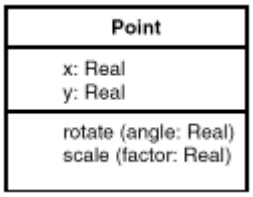
2nd exemple de contrainte : un employé et son chef doivent travailler dans la même entreprise (`{Person.employer=Person.boss.employer}`).

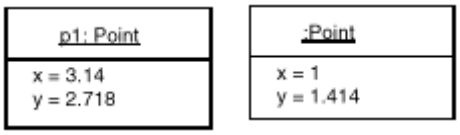


- propriété (*property*) : valeur nommée représentant une caractéristique d'un élément de modélisation
  Format d'une propriété : ◁`nom propriété`▷ = ◁`valeur`▷
  Exemples de propriétés :
  ```
  { author = "Joe Smith", deadline = 31-March-1997, status = analysis }
  { abstract }
  ```
- valeur marquée (*tagged value*) : définition explicite d'une propriété en précisant ses nom/valeur
- commentaire (*comment*) : chaîne attachée à un élément de modélisation, mais sans sémantique
  Exemples de commentaires :
  ```
  flight.pilot.training_hours > flight.plane.minimum_hours
  company.employees->select (title = "Manager" and self.reports->size > 10)
  ```
- note (*note*) : information textuelle (commentaire bien évidemment mais aussi contrainte, valeur marquée, corps d'une méthode ou d'autres chaînes valuées) associée à un (ou plusieurs) élément(s) d'un modèle
  Exemple de note :

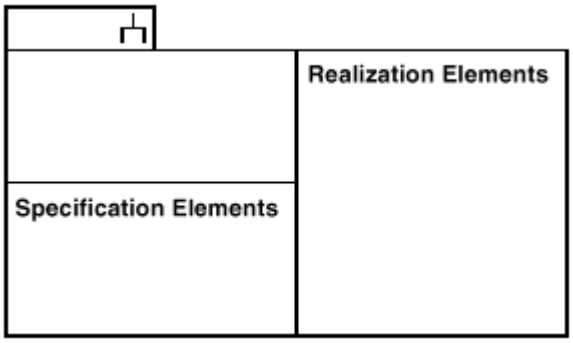


- dépendance (*dependency*) : relation d'obsolescence (unidirectionnelle) entre deux éléments de modélisation
- dichotomie type/instance (*type-instance correspondence*) : séparation de l'essence de l'élément de modélisation (sa spécification) et de la manifestation avec ses valeurs (sa réalisation) de ce type ; cela concerne la séparation classe/objet, association/lien, paramètre/valeur, opération/appel, etc.
  Exemple de dichotomie classe/objet : la classe `Point` (à gauche) et deux objets `p1:Point` et `:Point` de cette classe (à droite).

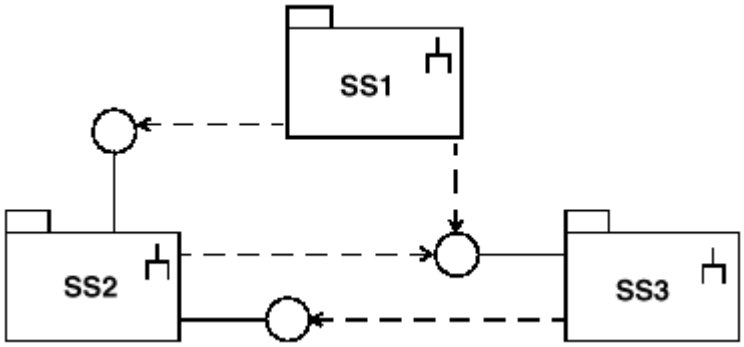




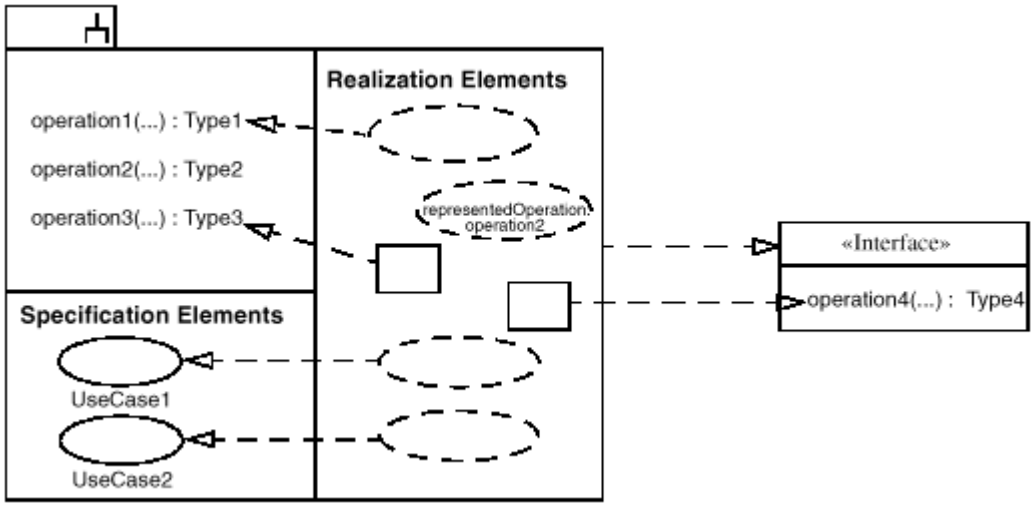
- dichotomie type/classe (*type/class dichotomy*) : séparation de la spécification d'un élément de modélisation et de la réalisation de cette spécification
- sous-système (*subsystem*) : ensemble d'éléments de modélisation groupés qui constituent une spécification du comportement présenté par les autres éléments du modèle
  Format d'un sous-système, avec ses trois compartiments (tous facultatifs) pour les opérations (en haut à gauche), les spécifications (en bas à gauche) et la réalisation (à droite).



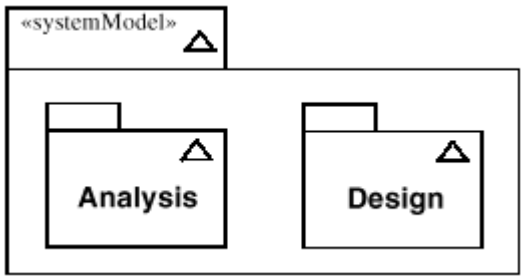
1er exemple de sous-système : trois sous-systèmes avec leurs interfaces et leurs dépendances.



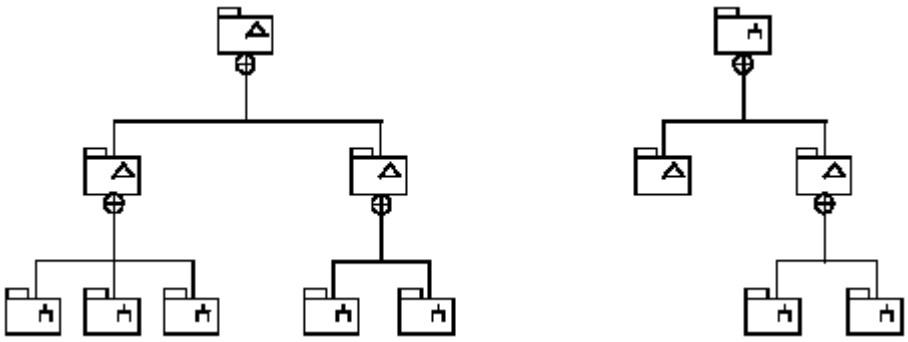
2nd exemple de sous-système : les trois compartiments sont présents et montrent les liens qu'il y a entre eux.



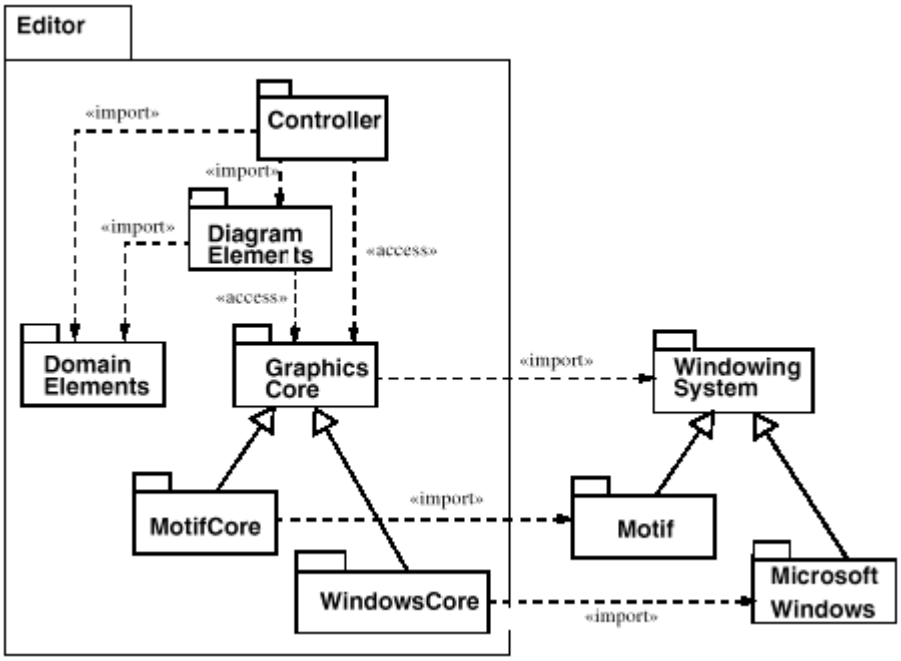
- modèle (*model*) : abstraction (sémantiquement fermée) d'un système (physique), contenant des éléments de modélisation
1er exemple de modèle : le modèle `«systemModel»` se compose d'un modèle d'analyse (`Analysis`) et d'un modèle de conception (`Design`).



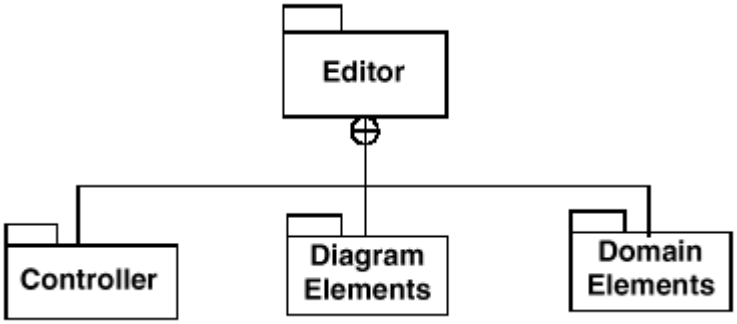
2ème et 3ème exemple de modèles : hiérarchies de modèles et de sous-systèmes (basé à gauche sur un modèle et à droite sur un sous-système).



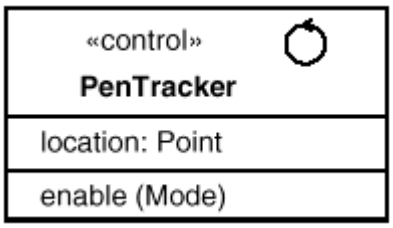
- paquetage (*package*) : regroupement (avec encapsulation) d'éléments de modélisation, selon un critère logique (et non fonctionnel)
  1er exemple de paquetage : plusieurs paquetages avec leurs relations d'accès («`access`») et d'import («`import`»).



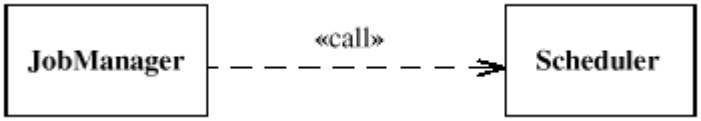
  2nd exemple de paquetage : le paquetage `Editor` est présenté sous la forme d'un arbre (composé de trois autres paquetages).



- stéréotype (*stereotype*) : extension de la sémantique d'un élément du méta-modèle
  Format d'un stéréotype : « ◁`stéréotype`▷ » (ou par une icône).
  1er exemple de stéréotype : le stéréotype `control` (à gauche) permettant de représenter un stylo (*PenTracker*) peut être simplifié soit en enlevant le texte «`control`», soit en enlevant le symbole (cercle fléché), soit en optant pour le dessin de droite.



  2nd exemple de stéréotype : le stéréotype `call` (le gestionnaire de tâches fait appel à un planificateur).



Le système est composé d'une hiérarchie de paquetages et d'un réseau de dépendances entre ces paquetages. En effet, les paquetages fournissent un mécanisme général pour la partition et le regroupement des modèles.
Les trois mécanismes d'extension sont : les stéréotypes, les valeurs marquées et les contraintes.

3. La modélisation

Ce chapitre présente les vues proposées par UML, les modèles et quelques informations pour utiliser les modèles selon le niveau d'abstraction.

3.1. Les vues

UML propose différents modèles pour représenter les différents points de vue de la modélisation.
Les 4+1 vues sont :

- la vue **logique** (intégrité de conception) : perspective abstraite de la solution (classes, relations, machines états-transitions, etc.) ;
- la vue des **composants** (intégrité de gestion du code) : perspective physique de l'organisation du code (modules, composants, concepts du langage ou de l'environnement d'implémentation) ;
- la vue des **processus** (intégrité d'exécution) : perspective sur les activités concurrentes et parallèles (tâches et processus) ;
- la vue de **déploiement** (intégrité de performance) : répartition du système (logiciel) à travers un réseau ;
- la vue des **cas d'utilisation** (intégrité de conception), qui guide et justifie les autres : moyen rigoureux et systématique pour guider la modélisation (cas d'utilisation, scénarios, collaborations d'objets, machines à états).

3.2. Les modèles

Les 9 modèles sont :

- diagramme de **classes** (*class diagram*) [Cf. OMT, Booch et autres méthodes objet] : structure statique (classes et associations) ;
- diagramme d'**objets** (*object diagram*) : instance du diagramme de classes (objets et liens) ;
- diagramme de **cas d'utilisation** (*use case diagram*) [Cf. OOSE] : fonctions du système du point de vue des utilisateurs ;
- diagrammes de comportement (*behavior diagrams*) :
  - diagrammes d'interaction (*interaction diagrams*) : interactions entre les objets (scénarios et flots de messages)
    - diagramme de **séquence** (*sequence diagram*) [Cf. divers modèles (*interaction, message trace, event trace*) d'anciennes méthodes non orientées objet] : aspect temporel des interactions entre les objets (séquence d'événements) ;
    - diagramme de **collaboration** (*collaboration diagram*) [Cf. notamment Booch (*object diagram*) et Fusion (*object interaction graph*)] : aspect spatial des interactions entre les objets (relativement à leurs rôles et aux liens qu'ils ont entre-eux) ;
  - diagramme d'**activités** (*activity graph diagram*) [Cf. diagrammes de flots de données (*work flow diagrams*) émanant d'anciennes méthodes (objet ou non)] : comportement d'une opération en terme d'actions et d'activités ;
  - diagramme d'**états-transitions** (*statechart diagram*) [Cf. travaux de D. Harel] : comportement dynamique des objets (changeant d'état en réaction à des événements) ;
- diagrammes de réalisation (*implementation diagram*) [Cf. Booch (*module* et *process diagram*)] : unités de travail
  - diagramme de **composants** (*component diagram*) : composants logiciels réalisant l'application (code source, bibliothèques, dépendances, etc.) ;
  - diagramme de **déploiement** (*deployment diagram*) : répartition des composants logiciels sur des matériels.

De plus, les stéréotypes [nouveauté d'UML] fournissent l'un des mécanismes d'extension, utilisés notamment pour étendre la sémantique du méta-modèle, tandis que le langage d'expression de contrainte objet OCL (*Object Constraint Language*) [Cf. Syntropy ou Catalysis] est utilisable durant toutes les phases du cycle de développement du logiciel, également utilisé par UML pour spécifier sa sémantique.

3.3. Utilisation des modèles

Rappelons qu'il faut dans un premier temps modéliser le **métier** de l'organisation étudiée, en enchaînant les étapes suivantes :

- étude du périmètre et des intervenants extérieurs à l'organisation,
- étude des processus de l'organisation,
- étude des travailleurs et des entités de l'organisation,
- étude des flots d'événements des processus,
- étude des structures organisationnelles.

Un **acteur** est une abstraction extérieure au système à modéliser (l'organisation lorsque l'on modélise le métier, le système informatique dans un second temps) qui interagit avec lui.
Il s'agit de rôles joués par des personnes, de logiciels, de matériels, etc.
On détermine un acteur principal en se demandant « À qui est destiné le système à modéliser ? ».
Un **processus** d'une organisation est soit un processus métier (visible de l'extérieur du système), soit un processus support (interne à l'organisation et donc non visible de l'extérieur, support aux processus métier pour s'exécuter), soit un processus de gestion (interne à l'organisation et donc non visible de l'extérieur, moyens de gestion des processus métier).
Un **processus métier** est l'ensemble des activités internes d'un métier permettant de fournir un résultat observable et mesurable pour un utilisateur individuel du métier.
Un processus métier est représenté par un cas d'utilisation (i.e. dans un diagramme de cas d'utilisation), et inversement.

Un **diagramme de cas d'utilisation** est un graphe d'acteurs, un ensemble de cas d'utilisation englobés par la limite du système, des relations (ou associations) de communication (participation) entre les acteurs et les cas d'utilisation, et des généralisations de ces cas d'utilisation.
Les relations de généralisation/spécialisation entre acteurs permettent de définir des profils d'acteurs. Les relations entre cas d'utilisation sont soit `utilise` (*uses*) pour éviter de dupliquer des processus communs à plusieurs processus, soit `étend` (*extends*) pour décrire séparément des parties alternatives ou optionnelles ou exceptionnelles de processus. Les relations entre acteurs et cas d'utilisation indiquent les interactions.
Un diagramme de cas d'utilisation ne détaille pas l'interaction entre acteur et processus (uniquement la relation et le sens du stimulus sont indiqués) car c'est l'objet de la description du cas d'utilisation.
Une organisation est un système métier dans lequel des **travailleurs** interagissent, communiquent et travaillent ensemble pour exécuter les processus métier. Ainsi, en utilisant les objets et les classes pour représenter les travailleurs du métier, on définit une organisation orientée objet.
Il existe trois stéréotypes prédéfinis pour les classes définies lors de la modélisation du métier : `travailleur d'interface` (*interface worker*) i.e. un travailleur dans le métier visible du point de vue d'un acteur, `travailleur interne` (*internal worker*) i.e. un travailleur dans le métier non visible du point de vue d'un acteur, et `entité` (*entity*) i.e. une **entité** manipulée dans le métier par les travailleurs (exemple : réglementation, procédure ; contre-exemple : matériel).
La description (textuelle ou via un **diagramme d'états-transitions**) d'un processus détaille les activités internes du processus, i.e. son **flux d'événements** (*workflow*), afin d'expliquer aux intervenants leurs rôles et activités ; la description textuelle comporte notamment une brève description du processus, un glossaire des termes du domaine, et une présentation de chaque flux d'événements normal et de chaque flux d'événements alternatif.
La description d'un flux d'événements d'un processus est utile pour montrer le détail des interactions entre les travailleurs, identifier les protocoles des travailleurs et des entités, décrire les parties complexes du processus, décrire exactement une séquence d'activités importantes, etc.
La description complète d'un flux d'événements d'un processus nécessite plusieurs diagrammes. Ainsi, le flux d'événements général est divisé en parties incomplètes représentées chacune par un diagramme d'interaction (diagramme de séquence et/ou diagramme de collaboration).
Le **diagramme de séquence** montre les interactions entre les objets, agencées en séquence dans le temps ; il montre en particulier les objets participant à l'interaction par leurs lignes de vie et les messages qu'ils s'échangent ordonnancés dans le temps … mais il ne montre pas les relations entre les objets.
Le **diagramme de collaboration** montre les interactions entre les objets et leurs liens ; il montre en particulier les relations entre les objets … mais il ne montre pas le temps.
Les diagrammes de séquence et de collaboration sont équivalents.
Le **diagramme d'activités** doit quant à lui décrire les activités (des travailleurs) du métier.
Le **diagramme de classes** permet de représenter des classes et leurs relations. Aussi, outre des classes, il peut contenir des types, des paquetages, des relations, voire des instances (objets et liens).
Différents types de relations peuvent être exprimées dans un diagramme de classes, et notamment l'association (relation bidirectionnelle entre plusieurs classes).
Le diagramme de classes exprime la traçabilité directe entre un processus, ses travailleurs et ses entités (i.e. entre le cas d'utilisation et ses classes). En ce sens, il s'agit d'un diagramme structurel statique.
L'architecture métier consiste, une fois les processus métier complètement décrits, à regrouper les travailleurs et les entités en **unités organisationnelles** par unités de compétence ou par structures organisationnelles (données par l'organigramme de l'organisation). Un travailleur appartient à un seul paquetage et, de même, une entité appartient à un seul paquetage ; par contre, une relation peut appartenir à plusieurs paquetages. Les stéréotypes suivants sont utilisés pour les paquetages : `organisation`, `entité externe` et `infrastructure`.
Le passage du métier au **système informatique** (comprenant les ressources humaines, les systèmes informatiques et d'autres ressources) se fait (plus ou moins) directement : les travailleurs métier deviennent des acteurs, les activités des travailleurs métier deviennent des cas d'utilisation (i.e. processus informatiques), les entités deviennent des objets entités. En plus des besoins fonctionnels ainsi obtenus, le système informatique devra considérer les besoins techniques, les aspects liés à la maintenance, les contraintes technologiques et politiques.
Aussi, dans un second temps, il faut modéliser le système informatique en enchaînant les étapes suivantes :

- étude du périmètre, des acteurs et des cas d'utilisation,
- description et structuration des cas d'utilisation,
- sélection et description des scénarios,
- identification des objets candidats,
- description des scénarios avec des diagrammes d'interaction,
- description des classes et de leurs relations,
- description de la dynamique de certaines classes.

L'objectif de la capture des besoins consiste à déterminer ce que le système doit faire. Ces besoins sont formalisés par un modèle de cas d'utilisation représenté par des acteurs et des cas d'utilisation.
Les activités **d'analyse et de conception** doivent décrire la manière dont le système réalise les besoins (décrits dans le modèle de besoins) et ainsi définir rapidement une architecture stable. Ainsi, l'analyse et la conception débouchent sur un modèle de conception basé sur le modèle de cas d'utilisation et sur le guide d'utilisation de l'environnement de développement. La conception contient la réalisation des cas d'utilisation décrite en terme de classes et d'objets participants.

Les perspectives à considérer pour l'analyse et la conception sont les vues logique (classes importantes, organisation des classes en paquetages et en sous-systèmes, organisation des sous-systèmes en couches, réalisation des cas d'utilisation), processus (tâches impliquées) et déploiement (nœuds physiques avec leurs tâches).
Les activités d'analyse et de conception couvrent trois aspects : architecture, cas d'utilisation, et objets.
Un mécanisme d'**architecture** est une (ou un ensemble de) classe(s) ou un pattern constituant une solution commune à un problème commun ; c'est un concept informatique et logiciel, et il n'est donc pas nécessairement lié au domaine.
Les mécanismes d'architecture sont de trois types : analyse (non contraint par l'environnement d'implémentation), conception (contraint par l'environnement d'implémentation : langage de programmation, progiciel, base de données, technologie de communication), et implémentation des mécanismes de conception.
Le périmètre du système informatique diffère de celui du métier. Aussi, les **acteurs** du système informatique sont les travailleurs du métier … mais pas les utilisateurs du métier (sauf s'ils ont directement accès au système informatique).
On détermine les acteurs du système (informatique) en répondant aux questions : « Quels sont les utilisateurs qui ont besoin du système pour réaliser leur travail ? », « Quels sont les utilisateurs qui exécutent les fonctions principales du système ? », « Quels sont les utilisateurs qui exécutent les fonctions secondaires (maintenance et administration) du système ? », « Est-ce que le système interagit avec du matériel et d'autres logiciels ? ».
Un **cas d'utilisation** définit le comportement d'un système ou la sémantique de toute autre entité en spécifiant une séquence d'actions (avec des variantes) que l'activité réalise en interagissant avec les acteurs de l'entité.
On détermine flux d'événements permettant d'identifier les cas d'utilisation candidats en répondant aux questions : « Quelles sont les tâches fondamentales que l'acteur veut faire faire au système ? », « Est-ce que l'acteur crée, modifie, supprime ou consulte des informations dans le système ? », « Est-ce que l'acteur a besoin d'informer le système de changements externes ? », « Est-ce que l'acteur a besoin d'être informé des occurrences survenues dans le système ? », « Est-ce que l'acteur commande le démarrage et l'arrêt du système ? ».
Un cas d'utilisation englobe plusieurs fonctions du système.
La description d'un cas d'utilisation peut être informelle (textuelle) ou plus formelle (diagramme d'états-transitions).
Deux descriptions présentent un cas d'utilisation : externe (du point de vue de l'acteur) et interne (du point de vue du système).
La description textuelle d'un cas d'utilisation comporte notamment une brève description et une présentation (sous la forme d'algorithmes généraux ou de pseudo-code) de chaque flux d'événements (normal ou alternatif).
La structuration du modèle de cas d'utilisation consiste à les regrouper en paquetages, à définir les variantes (flux normal, flux alternatifs – variantes, exceptions, cas d'erreurs – , extensions des cas normaux), et à définir des points communs (i.e. décrire dans un cas d'utilisation séparé un sous-ensemble commun utilisé par plusieurs cas d'utilisation).
La présentation du modèle de cas d'utilisation peut comprendre un diagramme de cas d'utilisation (acteurs, cas d'utilisation et relations) pour les acteurs appartenant au même paquetage de cas d'utilisation, pour un acteur et tous les cas d'utilisation avec lesquels il interagit, les cas d'utilisation utilisés par le même groupe d'acteurs, pour les cas d'utilisation souvent exécutés en séquence, pour un cas d'utilisation avec ses variantes et ses sous-cas d'utilisation, pour les cas d'utilisation les plus importants.
Un cas d'utilisation est une abstraction de plusieurs chemins d'exécution, et un **scénario** est une instance d'un cas d'utilisation.
Une fois les cas d'utilisation décrits, il faut sélectionner un ensemble de scénarios (pour chaque cas d'utilisation) qui serviront qui serviront à piloter l'itération en cours de développement. Le choix repose sur l'élimination des risques (les plus importants) … et donc commence par le plus difficile !
Les scénarios sont classés : principal (cas normal) ou secondaire (cas alternatif, exception, erreur).
Les scénarios servent à analyser et concevoir le système, justifier a posteriori les choix d'architecture, et tester le logiciel.
La description textuelle d'un scénario (*script*) permet d'identifier les objets candidats (et les interactions). Ensuite, le scénario peut être représenté graphiquement par un ou plusieurs diagrammes d'interaction (diagrammes de séquence et/ou diagrammes de collaboration) basés sur les objets candidats précédemment identifiés ; il est cependant possible d'introduire de nouveaux objets (formulaire de saisie servant d'interface entre l'acteur et le système, objet englobant plusieurs règles de gestion et s'occupant de leur enchaînement et de leur validation, etc.).
Une **classe** est un ensemble d'objets possédant une structure, un comportement et des relations similaires.
Une classe est statique (i.e. c'est une description) contrairement à un objet qui est actif.
On construit plusieurs diagrammes de classe qui montrent les classes qui participent à un cas d'utilisation, les classes qui participent à un scénario, les classes privées d'un paquetage, les classes publiques d'un paquetage, une hiérarchie de paquetages, le détail d'une classe avec ses relations principales.
Il existe trois stéréotypes prédéfinis pour distinguer les objets informatiques d'une classe : `frontière` (*boundary*) i.e. une classe dont les objets sont visibles par les acteurs du système, `entité` (*entity*) qui représente des phénomènes internes au système (souvent des objets persistants), et `contrôle` (*control*) i.e. une classe interne au système qui contrôle le comportement d'un ou plusieurs cas d'utilisation (exemples : activité de contrôle, règle de gestion, superviseur, moyen de découplage entre objets, etc.).
Le comportement d'une classe est décrit par l'ensemble de ses responsabilités qui sont elles-mêmes mises en œuvre par des **opérations**.
Une opération possède une signature (type de retour et liste des paramètres avec le type de chacun), peut avoir des pré ou des post-conditions, etc.
Les messages des diagrammes d'interaction qui arrivent sur une instance sont des candidats pour donner naissance à des opérations.
La difficulté consiste à trouver un équilibre, lors de leurs spécifications, entre opérations élémentaires ou complexes.
Un **attribut** est une propriété nommée de classe qui porte un ensemble de valeurs que les objets vont prendre.
Un attribut possède un type, une valeur initiale, et une multiplicité.

Afin de protéger la cohérences des données, il est conseillé de rendre privé tous les attributs, et donc de fournir des opérations (publiques) d'accès aux attributs.
On distingue différents types de relations entre classes : l'association (relation statique reliant plusieurs classes entre elles), l'agrégation (variante de l'association qui définit une relation `partie de` entre instances de classes), la généralisation (relation statique de classification entre une classe plus spécifique qui contient par héritage tous les attributs, membres, relations d'une classe plus générale), dépendance (relation unidirectionnelle entre deux classes qui ne requiert pas forcément des instances ; les différentes formes sont : la trace qui établit une relation historique entre deux éléments, le raffinement qui décrit une dérivation historique entre deux éléments avec une transformation, et l'utilisation), etc.
Une classe-association est une association qui est en même temps une classe.
Les **paquetages** possèdent les éléments de modélisation. Un élément peut être public ou privé.
Les relations entre paquetages sont l'appartenance stricte (imbrication des paquetages) et l'utilisation.
Quelques critères de regroupement techniques : grouper les classes sémantiquement proches, organiser les paquetages en couches (i.e. qu'un paquetage n'utilise que les paquetages de même niveau ou de niveau inférieur), éviter les couplages forts entre paquetages, etc.
Les paquetages permettent de structurer le modèle, la réutilisation, d'organiser l'équipe de développement (pour répartir le travail et les responsabilités), de jeter les bases de la gestion de la configuration.
Une **interface** est la déclaration d'une collection d'opérations utilisables pour définir un service.
Les relations sur les interfaces sont : la fourniture (une classe donnée fournit l'interface à ses clients), l'utilisation (toute classe cliente qui désire utiliser l'interface), et la réalisation (une classe fournit une implémentation des opérations définies dans une autre classe).
Une **machine à états** est un comportement qui spécifie les suites d'états qu'un objet ou une interaction traverse durant sa vie en réponse à des événements, avec les réponses et les actions.
Une machine à états est un automate fini, déterministe et connexe.
Un état est une situation stable dans laquelle un objet peut être observé en train d'exécuter une activité (ou attendre un événement). L'état initial (*start*) est obligatoire et unique tandis qu'il peut y avoir aucun, un ou plusieurs états finaux (*stop*).
Un événement est une circonstance à laquelle un objet est susceptible de réagir. Les types d'événements sont : appel (réception d'un requête pour demander l'exécution d'une opération), changement (de valeur, d'attribut ou d'une relation), signal (en réception), et temporel (résultat d'une échéance temporelle).
Une transition est une relation entre un état source et un état cible, survenant soit suite à un événement, soit automatiquement.
Une condition de garde permet de définir si la transition à effectuer est valide ou non (i.e. que la transition est franchie uniquement si l'expression est vraie).
Une action est un traitement atomique exécuté au cours d'une transition. De plus, les événements d'entrée (*entry*) dans un état ou de sortie (*exit*) d'un état peuvent donner lieu à une action.
Une activité est un traitement effectué par un objet dans un état.
On peut créer une hiérarchie d'états : un super-état se décompose en sous-états (qui possèdent leurs propres machines à états).
Les **diagrammes d'états-transitions** sont des machines à états.
On ne définit un diagramme d'états-transitions pour un classe que si sa dynamique est complexe.
Le **diagramme d'activité** est une machine à états particulière dans laquelle les états sont des activités représentant la réalisation d'opérations et où les transitions sont déclenchées par la fin des opérations.
Un diagramme d'activité est attaché dans sa globalité à une classe ou à l'implémentation d'une opération ou encore à un cas d'utilisation.
Un état action est un état qui contient une action interne et au moins une transition sortante.
Un état action peut déclencher plusieurs transitions qui doivent être accompagnées chacune d'une condition de garde toutes mutuellement exclusives (concept de décision).
Un diagramme d'activité peut être découpé en couloirs représentant chacun une responsabilité de l'activité et assignée à un objet (concept de partition).
Les **stéréotypes** introduisent une distinction d'usage.
Un stéréotype est associable à tout élément de modélisation (classe, opération, attribut, relation, paquetage, composant, nœud, etc.).
Le langage d'expression de contraintes **OCL** permet de spécifier des invariants sur les classes dans le modèle de classes, de spécifier des invariants pour les stéréotypes, de décrire des pré et post-conditions sur les opérations, de décrire des conditions de garde, de servir de langage de navigation, de spécifier les contraintes sur les opérations.

4. L'exemple « jouet »

Le jour de la rentrée, le secrétariat de l'établissement avise les étudiants qu'ils ont jusqu'à la fin de la semaine pour amener les originaux des diplômes qu'ils ont obtenus, ceci permettant de compléter les fiches des étudiants (qui figurent ci-après).

Numéro INE de l'étudiant : 2
Nom : LEROI
Département de naissance : 40

Voitures possédées par l'étudiant :

Numéro d'immatriculation	Couleur

Diplômes obtenus par l'étudiant :

Intitulé abrégé	Intitulé complet	Année
BAC	Baccalauréat	1980
DEUG	Diplôme d'Études Universitaires Générales	1982

Numéro INE de l'étudiant : 3
Nom : DUPOND
Département de naissance : 17

Voitures possédées par l'étudiant :

Numéro d'immatriculation	Couleur

Diplômes obtenus par l'étudiant :

Intitulé abrégé	Intitulé complet	Année
BAC	Baccalauréat	1981
DUT	Diplôme Universitaire de Technologie	1983
MIAGe	Maîtrise des Méthodes Informatiques Appliquées à la Gestion des Entreprises	1985

Numéro INE de l'étudiant : 4
Nom : MARTIN
Département de naissance : 47

Voitures possédées par l'étudiant :

Numéro d'immatriculation	Couleur
4747 LA 47	rouge

Diplômes obtenus par l'étudiant :

Intitulé abrégé	Intitulé complet	Année
BAC	Baccalauréat	1977
DEUG	Diplôme d'Études Universitaires Générales	1980
MIAGe	Maîtrise des Méthodes Informatiques Appliquées à la Gestion des Entreprises	1982

Numéro INE de l'étudiant : 5
Nom : DURAND
Département de naissance : 33

Voitures possédées par l'étudiant :

Numéro d'immatriculation	Couleur
3333 BX 33	rouge
4040 NT 40	jaune

Diplômes obtenus par l'étudiant :

Intitulé abrégé	Intitulé complet	Année
BAC	Baccalauréat	1981
DUT	Diplôme Universitaire de Technologie	1983

Numéro INE de l'étudiant : 7
Nom : LEROI
Département de naissance : 33

Voitures possédées par l'étudiant :

Numéro d'immatriculation	Couleur

Diplômes obtenus par l'étudiant :

Intitulé abrégé	Intitulé complet	Année

5. Les modèles

Ce chapitre décrit les neuf diagrammes d'UML : de classes, d'objets, de cas d'utilisation, de séquence, de collaboration, d'activités, d'états-transitions, de composants et de déploiement.

5.1. Diagramme de classes

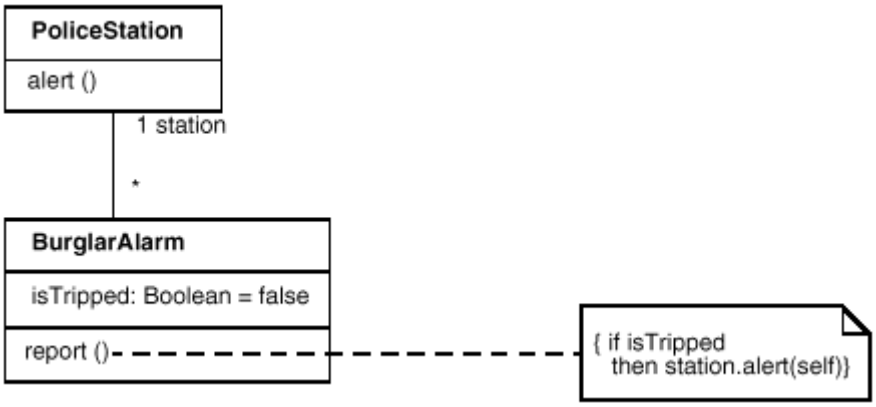
Un **objet** (*object*) est une entité ayant une frontière et une identité bien définie.
Un **attribut** (*attribute*) est une caractéristique d'un objet.
La syntaxe d'un attribut : ◁`visibilité`▷ ◁`nom attribut`▷ `[` ◁`multiplicité`▷ `]` `:` ◁`type retour`▷ `=` ◁`valeur initiale`▷ `{` ◁`propriété`▷ `}` où la visibilité est soit `+` (public), soit `#` (protégé), soit `-` (privé) et dont la multiplicité est de `1..1` par défaut.
Un attribut souligné correspond à un attribut de classe.
Exemples d'attributs : l'attribut `size` est public, de type `Area` définissant un carré de `100` de côté (non modifiable) ; l'attribut `colors` doit prendre exactement trois valeurs ; l'attribut `name` est facultatif (il peut prendre au plus une valeur).

```
+size: Area = (100,100) { frozen }
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindowPtr
colors [3]: Color
points [2..*]: Point
name [0..1]: String
```

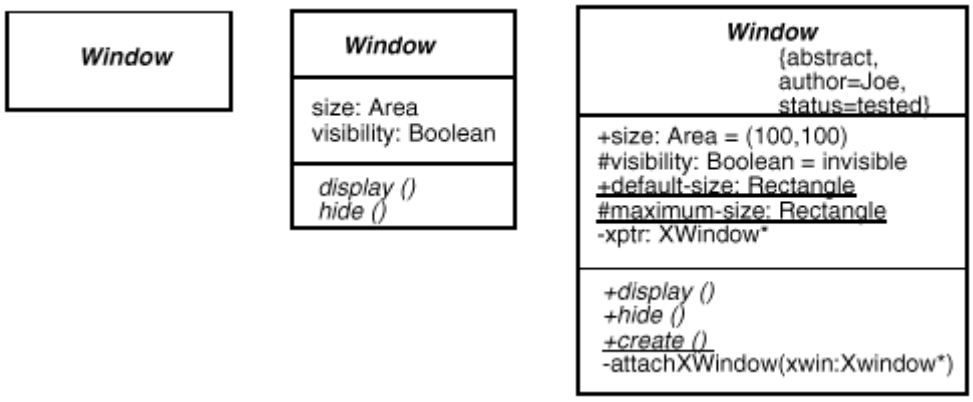
Une **opération** (*operation*) est un service dont un objet peut demander l'exécution.
La syntaxe d'une opération : ◁`visibilité`▷ ◁`nom opération`▷ `(` ◁`liste paramètres`▷ `)` `:` ◁`type retour`▷ `{` ◁`propriété`▷ `}` où la visibilité est soit `+` (public), soit `#` (protégé), soit `-` (privé) et un paramètre suit la syntaxe ◁`entrée/sortie`▷ ◁`nom paramètre`▷ `:` ◁`type`▷ `=` ◁`valeur défaut`▷ où l'entrée/sortie est soit `in` (par défaut), soit `out`, soit `inout`.
Exemples d'opérations : l'opération `attachXWindow` est privée et requiert le paramètre `xwin` de type `Xwindow*`.

```
+create ()
+display (): Location
-attachXWindow(xwin:Xwindow*)
```

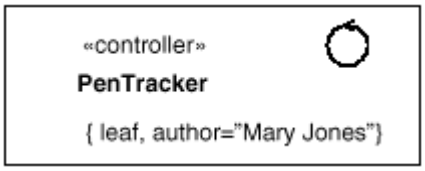
Le corps d'une opération peut être montré dans une note attachée à l'opération.
L'exemple ci-dessous montre le corps `if isTripped then station.alert(self)` de l'opération `report`.



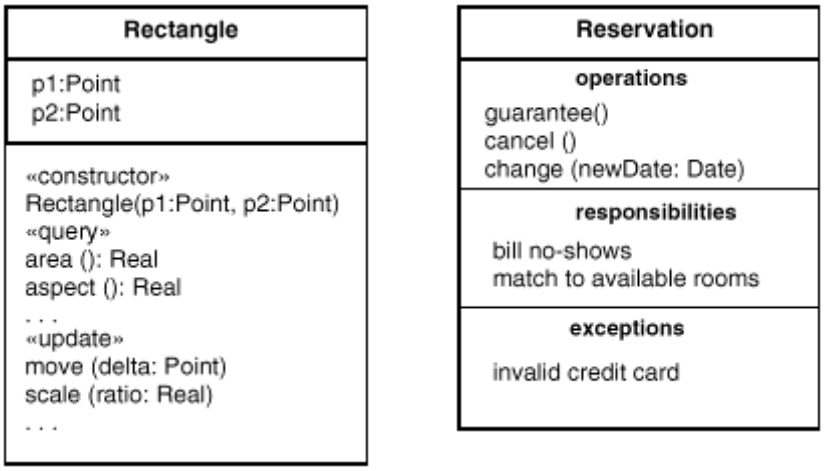
Une **classe** (*class*) est la description d'un ensemble d'objets ayant une structure (attributs), un comportement (opérations et méthodes) et des relations similaires.
Une classe est généralement représentée par un rectangle ayant trois compartiments : le nom de la classe (avec éventuellement un stéréotype et des propriétés), les attributs et les opérations. On peut simplifier la représentation en ne conservant que le premier compartiment. Inversement, on peut rajouter des compartiments (éventuellement nommés) pour les règles de gestion (*business rules*), les responsabilités, les événements, les exceptions, etc.
On peut également regrouper des listes d'éléments d'un compartiment en leur appliquant un stéréotype.
L'exemple ci-après illustre différentes représentations possibles de la classe `Window` : sans aucun détail (à gauche), au niveau de l'analyse (au centre) et au niveau de l'implémentation (à droite). Ainsi, comme l'illustre le dessin du centre, la classe `Window` a deux attributs (`size` et `visibility`) et deux opérations (`display` et `hide`).



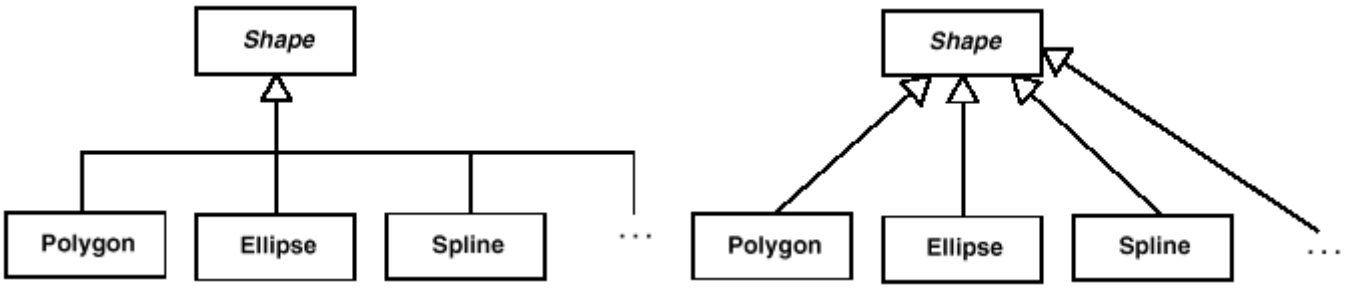
L'exemple ci-dessous illustre une classe (`PenTracker`) avec un stéréotype (`controller` et son symbole) et deux propriétés (`leaf` et `author`).



L'exemple ci-dessous illustre (à gauche, classe `Rectangle`) des stéréotypes (`constructor`, `query`, `update`) appliqués à des listes d'opérations et (à droite, classe `Reservation`) des compartiments nommés (`operations`, `responsibilities`, `exceptions`).



La **généralisation** (*generalization*) concernant les classes est une relation de taxinomie (classification) entre une classe générale (le père) et une classe spécifique (le fils) telle que d'une part la classe spécifique est totalement cohérente avec la classe générale et d'autre part la classe spécifique ajoute des informations à la classe générale.
Le symbole `...` indique que d'autres classes spécifiques existent dans le modèle mais ne figurent pas sur le schéma.
L'exemple ci-dessous (les deux représentations sont équivalentes) montre que la classe générale `Shape` (forme) se spécialise en trois classes spécifiques `Polygon`, `Ellipse` et `Spline`.

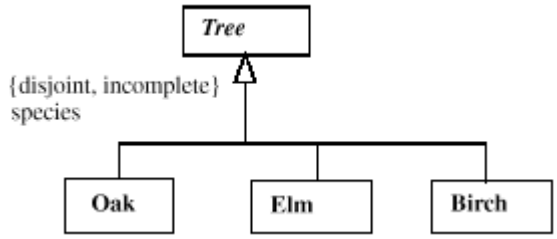


Le discriminant (*discriminator*) permet de regrouper différentes classes spécifiques.
Différentes contraintes peuvent être appliquées sur les classes générale/spécifiques :

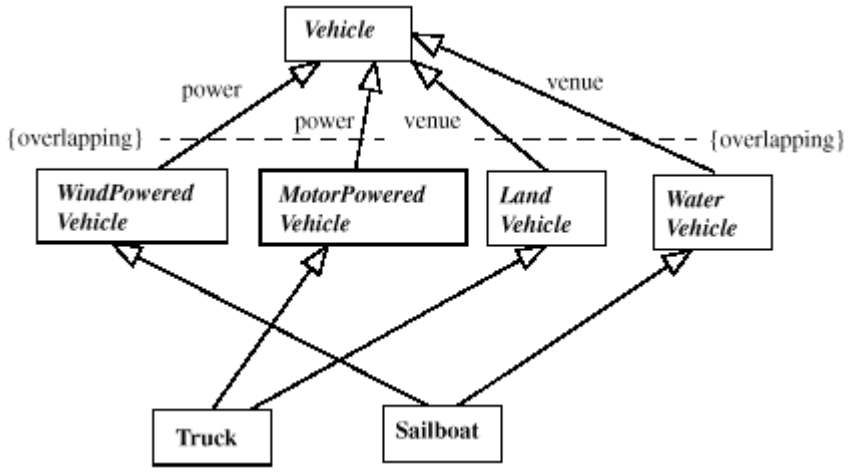
- chevauchement (`{overlapping}`) i.e. qu'un objet de la classe générale peut donner lieu à plusieurs objets des classes spécifiques,
  disjoint (`{disjoint}`) i.e. qu'un objet de la classe générale ne peut pas donner lieu à plusieurs objets des classes spécifiques ;

- complet (`{complete}`) i.e. que la liste de toutes les classes spécifiques est spécifiée,
  incomplet (`{incomplete}`) i.e. qu'on sait que la liste de toutes les classes spécifiques n'est pas entièrement spécifiée.

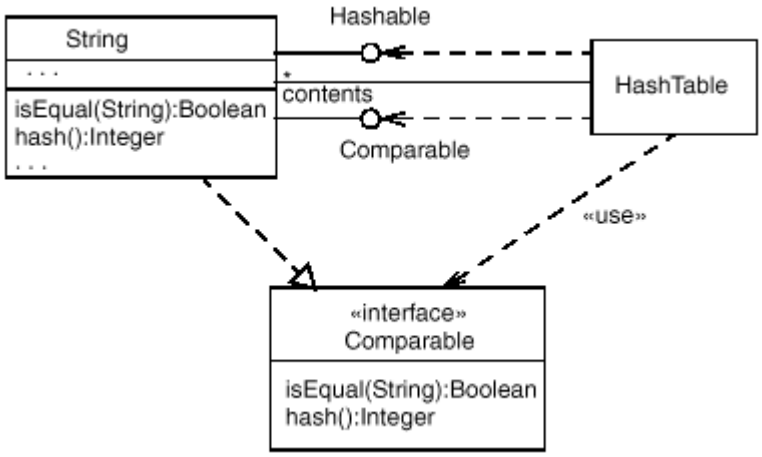
L'exemple ci-dessous montre que `Tree` (arbre) se spécialise en trois (mais il y en a d'autres) espèces disjointes `Oak` (chêne), `Elm` (orme) et `Birch` (bouleau).



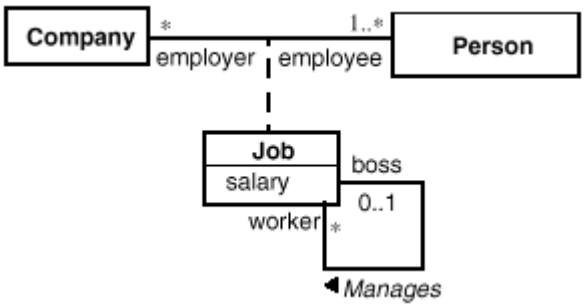
L'exemple ci-après montre que les véhicules (`Vehicle`) sont discriminés selon la puissance (`power`) et selon le lieu d'utilisation (`venue`). Pour les deux discriminants, le chevauchement est possible ; par exemple, et selon la puissance, un avion ultra léger motorisé (ULM) peut voler soit grâce au vent, soit avec son moteur. Quant au camion (`Truck`), c'est un véhicule (`Vehicle`) terrestre (`Land Vehicle`) équipé d'un moteur (`MotorPowered Vehicle`).



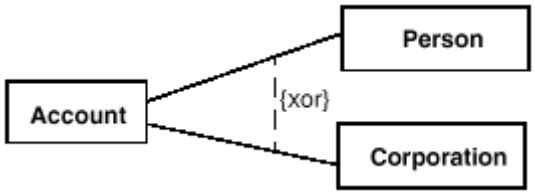
Une **interface** (*interface*) est la spécification d'opérations d'une classe visibles de l'extérieur.
L'exemple ci-dessous présente deux interfaces (`Hashable` et `Comparable`) pour la classe `String`, toutes deux utilisables par la classe `HashTable`. De plus, la liste des opérations (`isEqual` et `hash`) de l'interface `Comparable` est donnée.



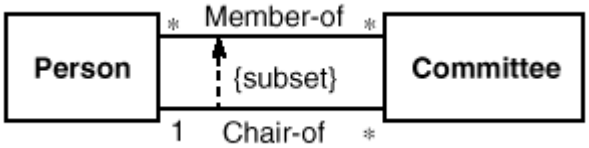
Un **lien** (*link*) est une connexion entre plusieurs objets.
Une **association** (*association*) est une relation statique entre plusieurs classes.
Une association binaire relie deux classes (éventuellement la même).
Une classe d'association (*association class*) est une association ayant des propriétés de classe … ou une classe ayant des propriétés d'association.
L'exemple ci-après montre l'association binaire `Job` (emploi) qui relie les classes `Company` (les employeurs) et `Person` (les employés), l'association `Manages` (dirige) qui est réflexive car elle relie deux fois la même classe `Job` (pour avoir les chefs et leurs subordonnés) et donc `Job` est une classe d'association.



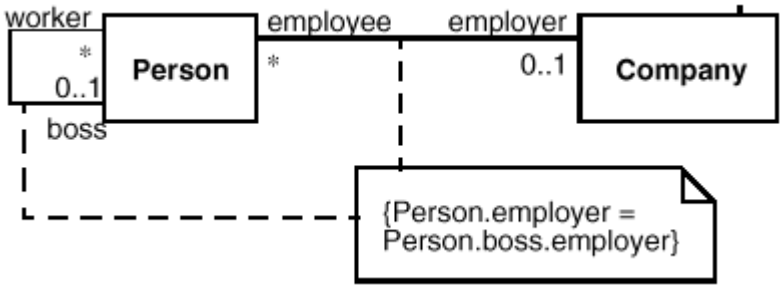
Un **rôle** (*role*) précise la signification de l'entité à proximité du rôle dans l'association.
Une **multiplicité** (*multiplicity*) spécifie l'ensemble des cardinalités possibles (parmi les entiers naturels) sur un rôle.
On peut préciser un intervalle (◁`début`▷`..`◁`fin`▷), une liste de valeurs (séparées par des virgules), ne pas borner supérieurement (`*`).
Exemples de multiplicités : `1` (un et un seul), `0..1` (un au plus : la valeur est facultative), `2..5` (de deux à cinq), `0..*` ou `*` (quelconque i.e. au moins zéro), `1..*` (au moins un), `2,5,7` (deux, cinq ou sept), `1,3..6,9..*` (un, de trois à six, ou au moins neuf).
Les liens ne sont pas triés par défaut (lorsque la multiplicité est strictement supérieure à un !) : pour indiquer que ces liens doivent être triés, on note la **contrainte** `{ordered}` sur le rôle correspondant.
D'autres contraintes peuvent être exprimées comme par exemple une contrainte d'exclusion (`xor-constraint`) entre associations indiquant que, pour chaque instance, une seule des associations concernées peut être instanciée à un moment donné.
L'exemple ci-dessous montre que chaque compte (`Account`) concerne soit une personne (`Person`), soit une société commerciale (`Corporation`).



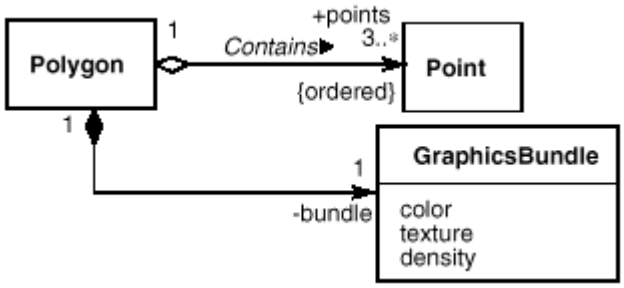
L'exemple ci-dessous montre que le président de chaque comité doit être l'un des membres de ce comité.



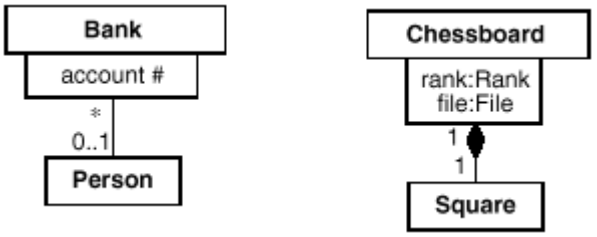
L'exemple ci-dessous montre qu'un employé et son chef doivent travailler dans la même entreprise.



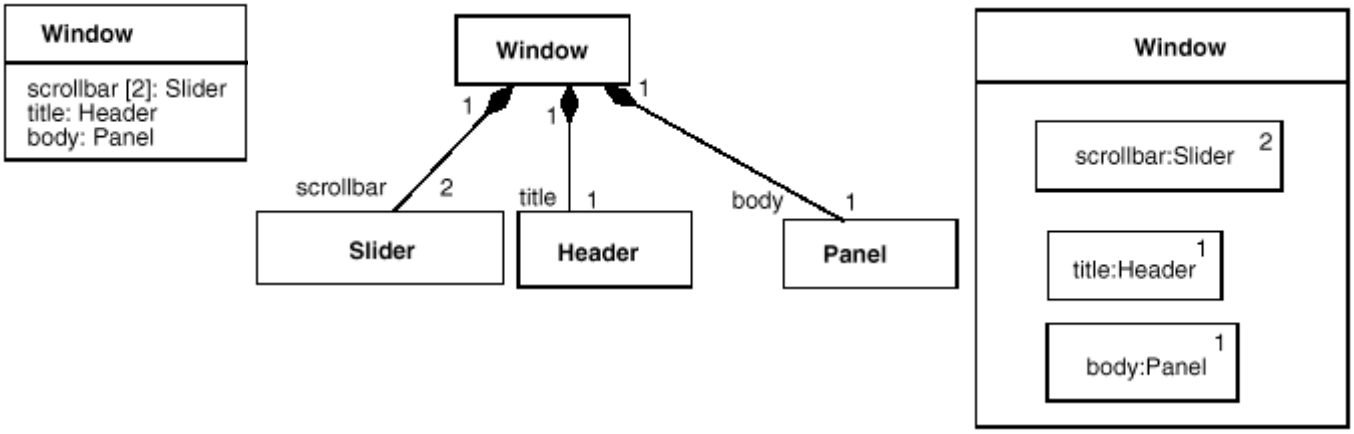
La navigabilité (*navigability*) indique que l'information ne peut être accédée via l'association que dans un seul sens (de la classe source vers la classe cible qui est pointée) ; par défaut, on peut accéder à une information dans n'importe quel sens.
La visibilité (*visibility*) attachée à un rôle peut être publique (`+` ou `{public}`), protégée (`#` ou `{protected}`) ou privée (`-` ou `{private}`).
La variabilité (*changeability*) indique si des données peuvent être mises à jour (ajoutées, modifiées, supprimées) ou non. Par défaut, une donnée peut être mise à jour. Par exemple, la contrainte `{frozen}` indique qu'une donnée, une fois créée, ne peut plus être mise à jour tandis que la contrainte `{addOnly}` indique qu'on ne peut qu'ajouter des données.
Il existe de nombreuses variantes d'associations : l'agrégation, la composition, la qualification.
L'**agrégation** (*aggregation*) est une relation « tout/partie » entre le composé (ensemble, agrégat) et le composant (l'élément).
La **composition** (*composition*) est une agrégation où la vie des composants est subordonnée à la vie de leurs composés.
La **qualification** précise un ensemble d'attributs (*qualifier*) dont les valeurs partitionnent l'ensemble des instances associées.
L'exemple ci-après montre deux associations : l'agrégation `Contains` qui relie les classes `Polygon` et `Point` (un polygone est composé d'une liste triée d'au moins trois points ; les points sont visibles ; on ne peut pas à partir d'un point retrouver le polygone) tandis que l'autre association est une composition.



Les deux exemples ci-dessous montre des associations qualifiées. Sur l'exemple de gauche, pour une banque et un numéro de compte donnés, il y a au plus une personne (tandis qu'une personne peut avoir aucun, un ou plusieurs comptes). Sur l'exemple de droite, un échiquier et un rang et un fichier donnés identifient une seule case.



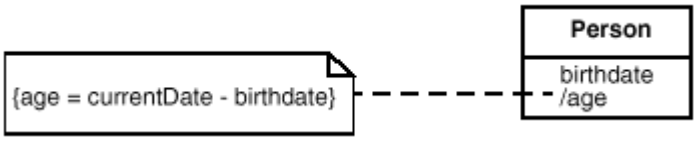
L'exemple ci-dessous montre de trois façons (différentes mais équivalentes) la composition d'une fenêtre en deux barres de défilement, un titre et un corps.



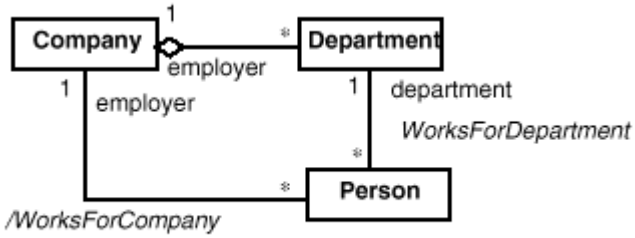
Une association **n-aire** (*n-ary association*) relie plus de deux classes (n est au moins égal à trois).
Pour la multiplicité d'associations n-aires, il convient de se référer au § 3.46.1 (p. 3-73) de la documentation : « Multiplicity for n-ary associations may be specified, but is less obvious than binary multiplicity. » ! Le modèle entités-associations est de ce point de vue beaucoup plus clair et sans ambiguïté (nombre d'occurrences à n-1 valeurs associées pour 1 occurrence de l'entité considérée). Fort heureusement, lorsqu'une association n-aire est justifiée, la cardinalité maximale est « plusieurs ».
L'exemple ci-dessous illustre une association (avec ses responsabilités) ternaire `Record` (enregistrement) qui relie les classes `Team` (équipe), `Year` (année) et `Player` (joueur) afin d'enregistrer que tel joueur était le gardien de but de telle équipe lors de telle saison.

Un **élément dérivé** (*derivated element*) est un élément (attribut, association, etc.) calculable à partir d'autres éléments.
La syntaxe d'un élément dérivé : `/ ◁nom élément▷`.

L'exemple ci-dessous illustre un attribut dérivé : l'âge (`age`) est calculé à partir de la date du jour (`currentDate`) et de la date de naissance (`birthdate`).

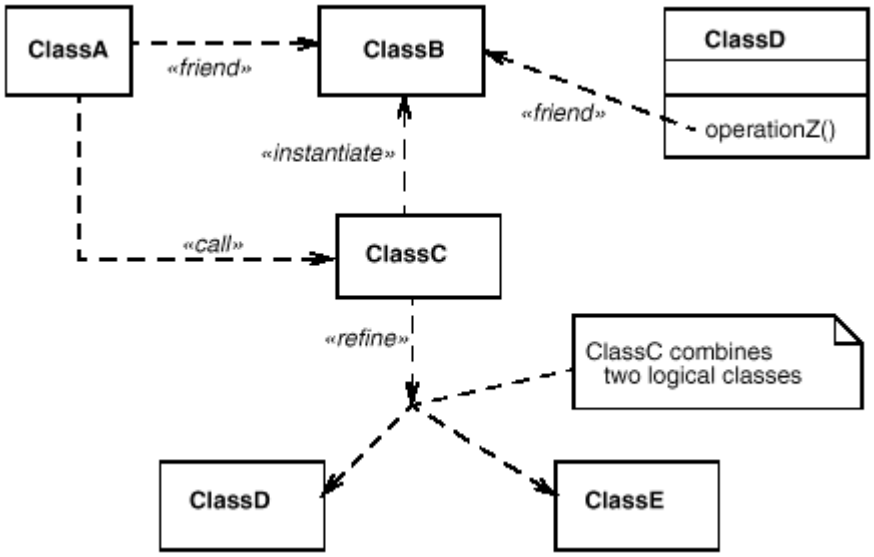


L'exemple ci-dessous illustre une association dérivée : on connaît l'entreprise pour laquelle une personne travaille (`WorksForCompany`) en passant par les deux associations reliant la classe des départements (`Department`) aux classes des personnes (`Person`) et des entreprises (`Company`).

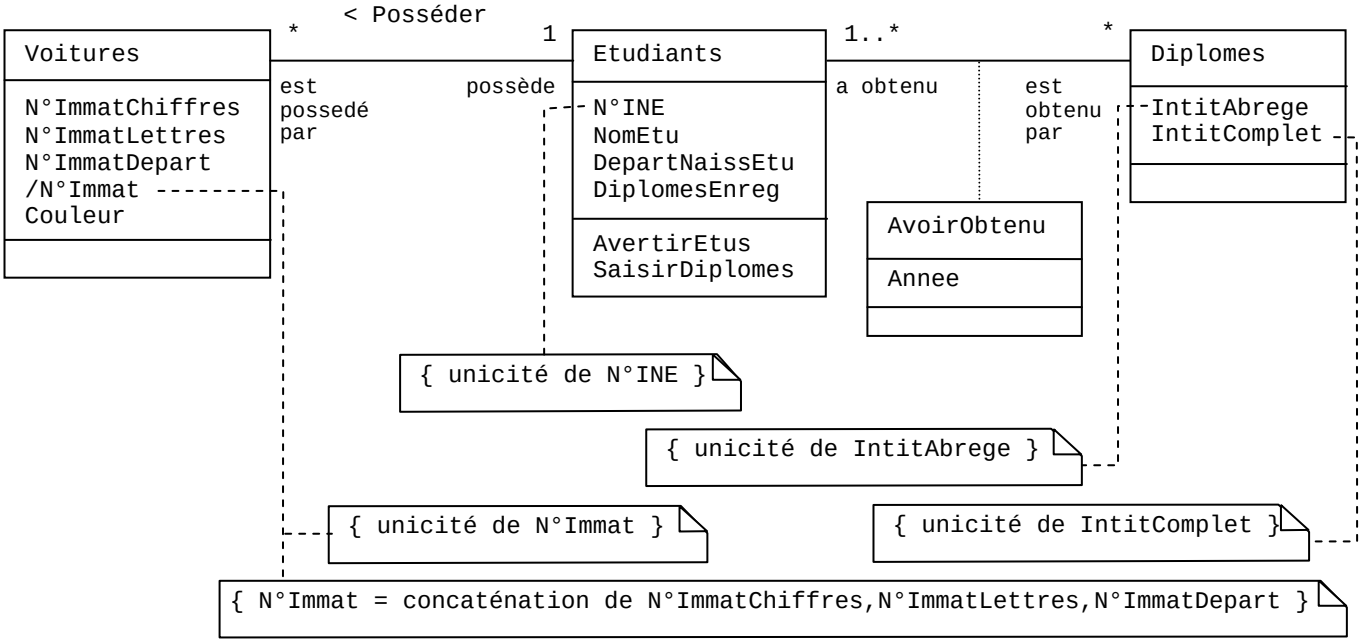


{ Person.employer=Person.department.employer }

Une **dépendance** (*dependency*) indique une relation entre deux classes (ou entre d'autres éléments de modélisation tel que les paquetages) où lorsqu'un changement intervient dans une classe, l'autre classe est affectée.
L'exemple ci-dessous illustre différentes sortes de dépendances (`friend`, `call`, `instantiate`, `refine`).

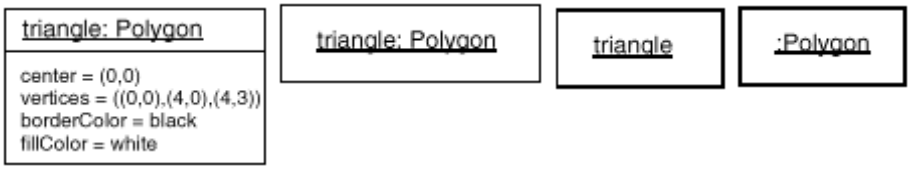


Un **diagramme de classes** (*class diagram*) montre la structure statique (et aucune information temporelle) d'un système.
Un diagramme de classes est représenté par un graphe d'éléments (classes, types, interfaces, paquetages, … et même instances) connectés par des relations.
Le diagramme de classes de l'exemple « jouet » (ci-après).


< Posséder
Voitures
N°ImmatChiffres
N°ImmatLettres
N°ImmatDepart
/N°Immat --------
Couleur
*
est
possedé
par
1
possède
Etudiants
N°INE
NomEtu
DepartNaissEtu
DiplomesEnreg
AvertirEtus
SaisirDiplomes
1..*
a obtenu
*
est
obtenu
par
Diplomes
IntitAbrege
IntitComplet
AvoirObtenu
Annee
{ unicité de N°INE }
{ unicité de IntitAbrege }
{ unicité de N°Immat }
{ unicité de IntitComplet }
{ N°Immat = concaténation de N°ImmatChiffres,N°ImmatLettres,N°ImmatDepart }

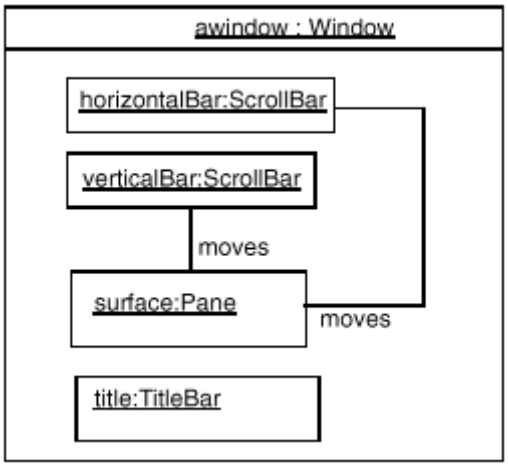
5.2. Diagramme d'objets

Un **objet** (*object*) est une entité ayant une frontière et une identité bien définies qui encapsule un état (attributs et relations) et un comportement (opérations, méthodes ou machines à états).
Un objet peut être représenté par un rectangle ayant deux compartiments : le nom d'objet et les attributs. Plus généralement, on simplifie la représentation en ne conservant que le premier compartiment.
La syntaxe d'un nom d'objet : ◁`nom objet`▷ : ◁`nom classe`▷ [◁`liste états`▷] où la classe peut préciser les paquetages d'imbrication successifs.
La syntaxe d'un attribut d'objet : ◁`nom attribut`▷ : ◁`type`▷ = ◁`valeur`▷.
L'exemple ci-dessous présente des triangles (les objets) instances de la classe des polygones.

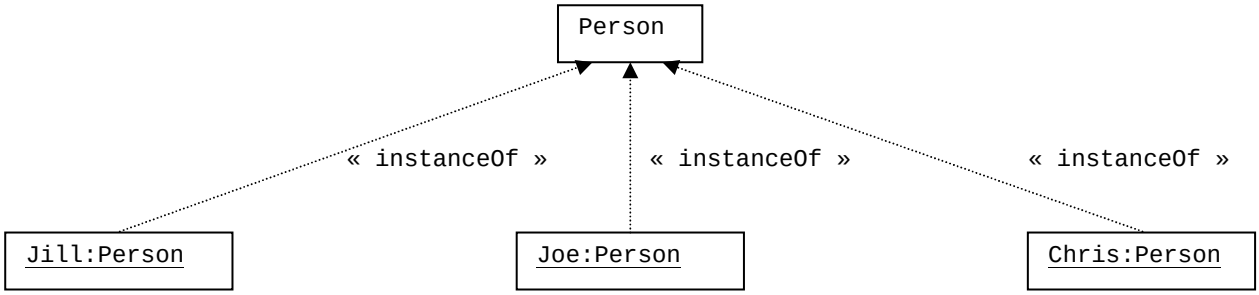


L'exemple ci-dessous illustre l'utilisation de l'icône d'un stéréotype pour l'objet `scheduler`.

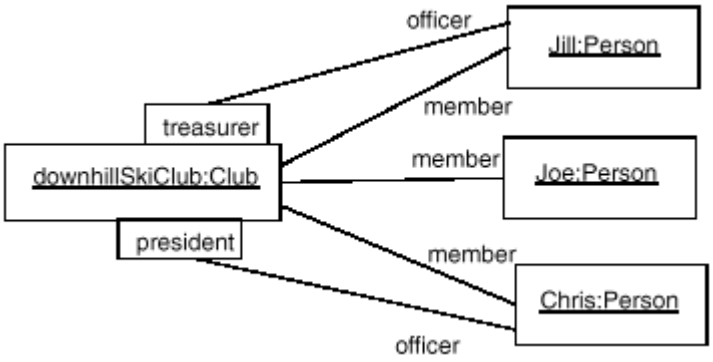
Un objet **composite** (*composite object*) est un objet de haut niveau constitué de parties fortement liées.
L'exemple ci-dessous illustre une fenêtre composée de deux barres de défilement, d'une surface et d'un titre.



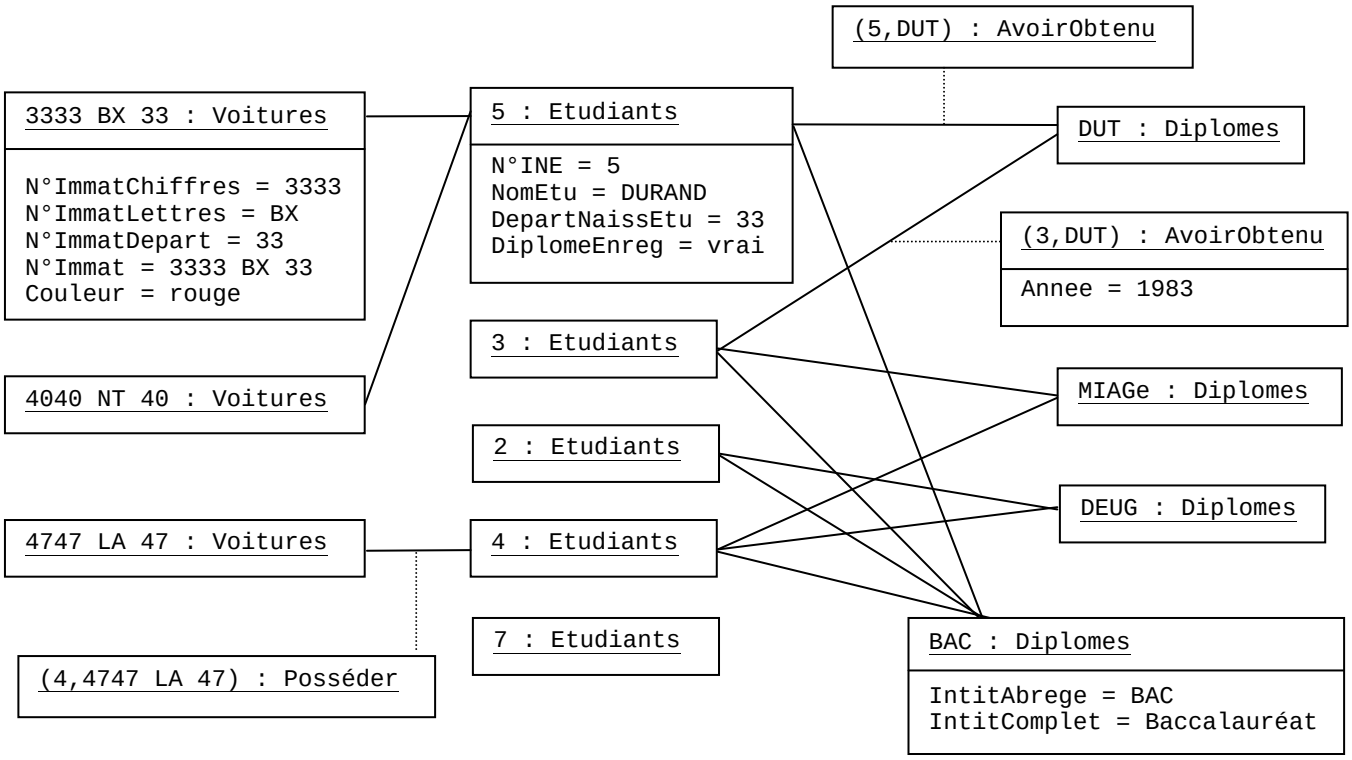
Une **classe** (*class*) est la description d'un ensemble d'objets ayant une structure, un comportement et des relations similaires.
Un objet est une instance d'une classe.
L'**instanciation** (*instanceOf*) montre la connexion entre une instance (*instance*) et son conteneur (*classifier*).
L'exemple ci-dessous montre l'instanciation de trois personnes.



Un **lien** (*link*) est une connexion entre plusieurs objets.
L'exemple ci-après montre un club de descente à ski (`downhillSkiClub`) composé de trois membres (`Jill`, `Joe` et `Chris`), et dont `Jill` est le trésorier (ou la trésorière) et `Chris` le président (ou la présidente). Clairement, le diagramme de classe correspondant comporte les deux classes `Club` et `Person` reliées par l'association `member` et par l'association `officer` qualifiée par la fonction exercée au sein du bureau du club.

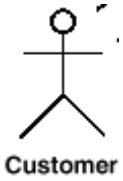


Une **association** (*association*) est une relation statique entre plusieurs classes.
Un lien est une instance d'une association.
Un lien est une liste de références d'objets (un couple de références dans le cas d'une association binaire).
Un **diagramme d'objets** (*object diagram*) montre des instances compatibles avec un diagramme de classes, à un moment donné.
Un diagramme d'objets est représenté par un graphe d'instances (objets et liens), avec les valeurs des données mais sans les classes.
Le diagramme d'objets de l'exemple « jouet » (ci-dessous). Les valeurs d'attributs de quelques objets (la voiture `3333 BX 33`, l'étudiant `5`, l'étudiant `3` ayant obtenu le diplôme `DUT`, le diplôme `BAC`) sont entièrement donnés.



5.3. Diagramme de cas d'utilisation

Un **acteur** (*actor*) est un ensemble cohérent de rôles joués par des entités externes (utilisateur, dispositif matériel ou autre système) qui interagissent avec le système.
Un acteur est représenté par l'icône stéréotype standard d'homme bâton (*stick man*) avec dessous le nom de l'acteur, ou par une classe avec le mot-clé `actor`.
L'exemple ci-dessous présente l'acteur `Customer` (client).

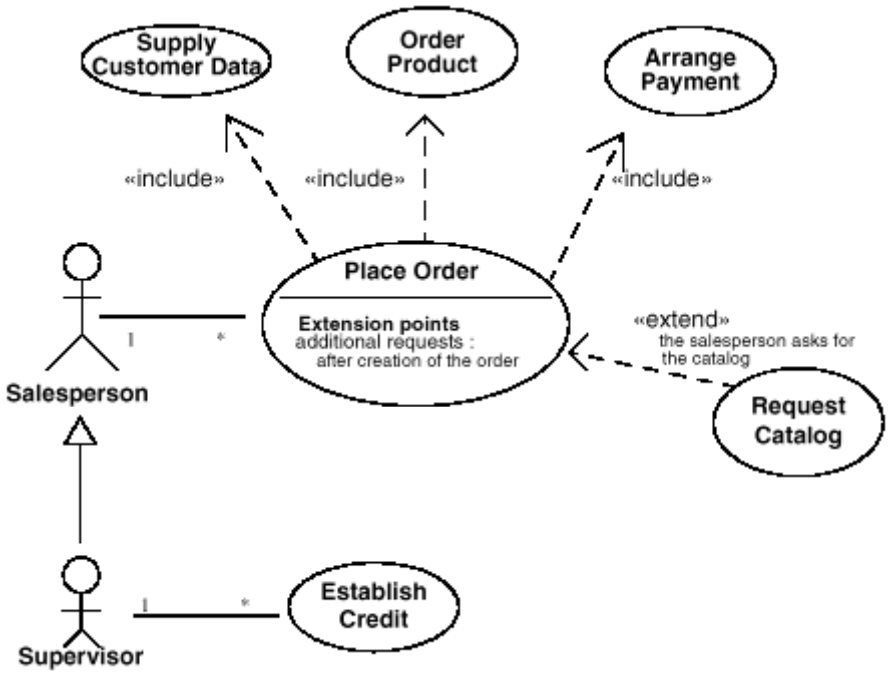


Un **cas d'utilisation** (*use case*) représente une unité cohérente d'une fonctionnalité fournie par un système (ou sous-système ou classe) spécifiée par une séquence d'actions que le système peut exécuter en interagissant avec les acteurs du système.
Un cas d'utilisation est représenté par une ellipse avec dedans le nom du cas d'utilisation, et éventuellement un stéréotype, des attributs, des opérations, et des points d'extension.
L'exemple ci-dessous présente le cas d'utilisation `Fill orders` (remplir les commandes).

Un point d'extension (*extension point*) référence un endroit dans le cas d'utilisation où la séquence d'actions d'un autre cas d'utilisation doit être insérée.
Les **relations** entre acteur et cas d'utilisation, entre acteurs, et entre cas d'utilisation sont les suivantes :

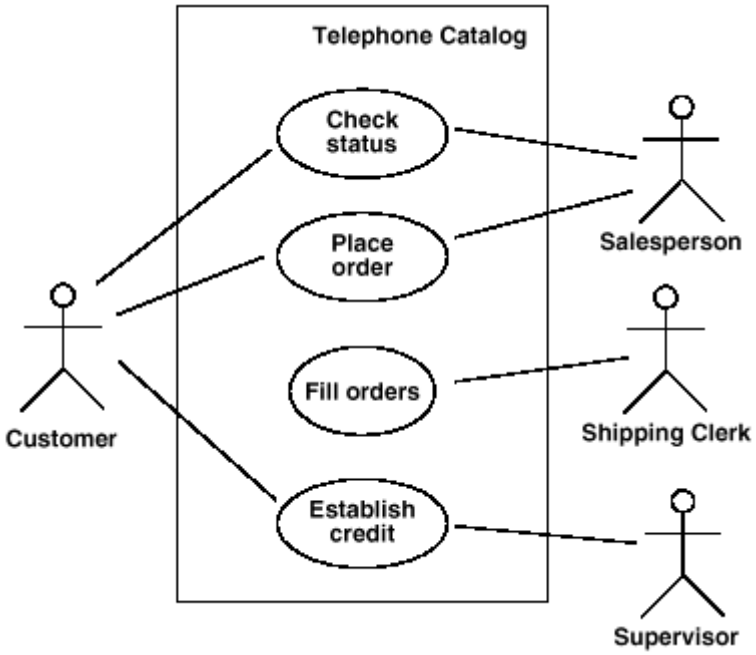
- l'association (*association*) exprimant la participation de l'acteur au cas d'utilisation ;
- la généralisation (*generalization*) d'un acteur A vers un acteur B indiquant qu'une instance de A peut communiquer avec les mêmes cas d'utilisation que les instances de B ;
- la généralisation (*generalization*) d'un cas d'utilisation A vers un cas d'utilisation B indiquant que A est une spécialisation de B ;
- l'extension (*extend*) d'un cas d'utilisation A vers un cas d'utilisation B indiquant qu'une instance de B peut être augmentée par le comportement spécifié dans A, à l'endroit défini par le point d'extension ;
- l'inclusion (*include*) d'un cas d'utilisation A vers un cas d'utilisation B indiquant qu'une instance de A contiendra le comportement spécifié dans B à un endroit défini.

L'exemple ci-dessous illustre notamment une association entre l'acteur `Salesperson` (vendeur) et `Place Order` (classer la commande), une généralisation de l'acteur `Supervisor` (chef de rayon) vers l'acteur `Salesperson` (vendeur), l'extension du cas d'utilisation `Request Catalog` (demander le catalogue) vers le cas d'utilisation `Place Order` (classer la commande), l'inclusion du cas d'utilisation `Place Order` (classer la commande) vers le cas d'utilisation `Order Product` (commander le produit).

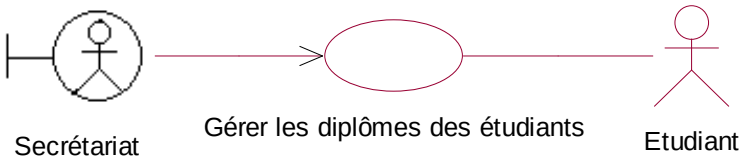


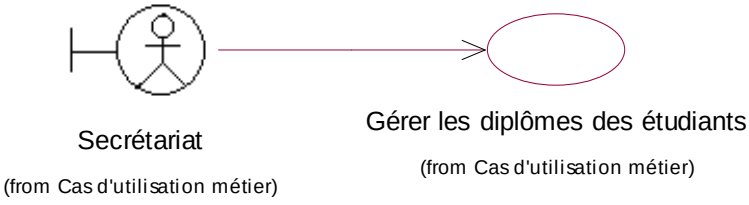
Un **diagramme de cas d'utilisation** (*use case diagram*) montre les relations entre les acteurs et les cas d'utilisation d'un système.

Un diagramme de cas d'utilisation est représenté par un graphe d'acteurs, un ensemble de cas d'utilisation, éventuellement des interfaces, et les relations entre ces éléments.
L'exemple ci-dessous présente le diagramme de cas d'utilisation `Telephone Catalog` (catalogue téléphonique) regroupant 4 acteurs et 4 cas d'utilisation. Ainsi, l'acteur `Shipping Clerk` (employé de bureau) est en relation (il s'agit d'une association) avec le cas d'utilisation `Fill orders` (remplir les commandes).



Remarque : dans un diagramme de cas d'utilisation, l'encadrement des cas d'utilisation (qui sert à montrer la frontière du système) est facultatif.
Les diagrammes de cas d'utilisation de l'exemple « jouet » (ci-dessous), du métier tout d'abord et du système informatique ensuite. Le travailleur d'interface `Secrétariat` utilise le cas d'utilisation `Gérer les diplômes des étudiants` qui concerne également l'acteur `Etudiant`.





5.4. Diagramme de séquence

Un **stimulus** (*stimulus*) est une communication entre deux objets qui s'échangent de l'information dans l'espoir qu'une action (exemples : invocation d'une opération, déclenchement d'un signal, création d'objet, suppression d'objet) s'ensuivra.
Un message (*message*) est la spécification d'un stimulus, précisant les rôles auxquels l'émetteur et le récepteur doivent se conformer.
Les différentes communications :

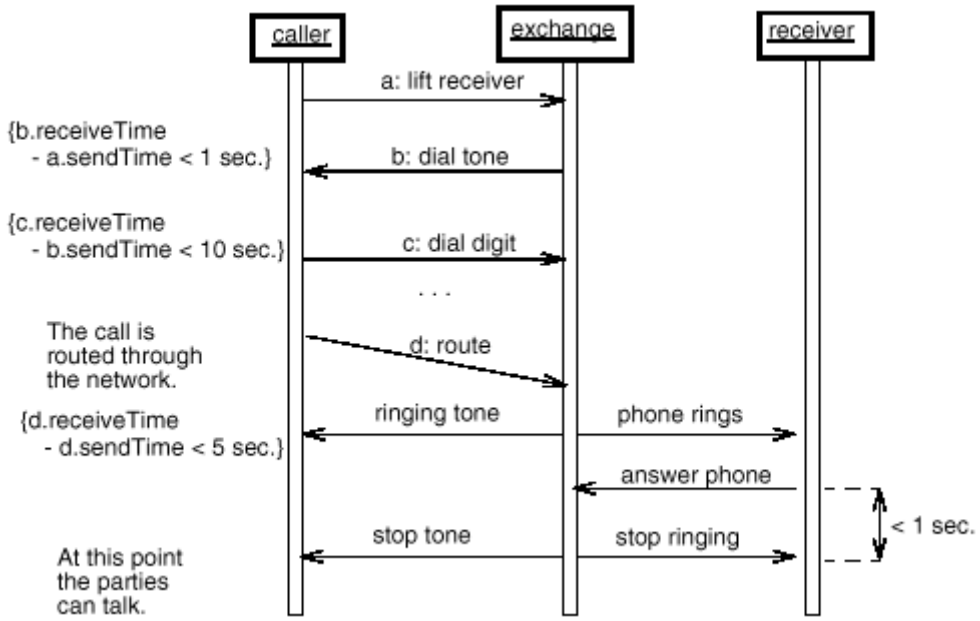
- appel de procédure (*procedure call*) ou flot de contrôle imbriqué (*nested flow of control*) : représentés par une flèche dont la pointe est remplie (*filled solid arrowhead*) ;

- flot de contrôle non imbriqué (*flat flow of control*) : représenté par une flèche dont la pointe n'est pas remplie (*stick arrowhead*) ;

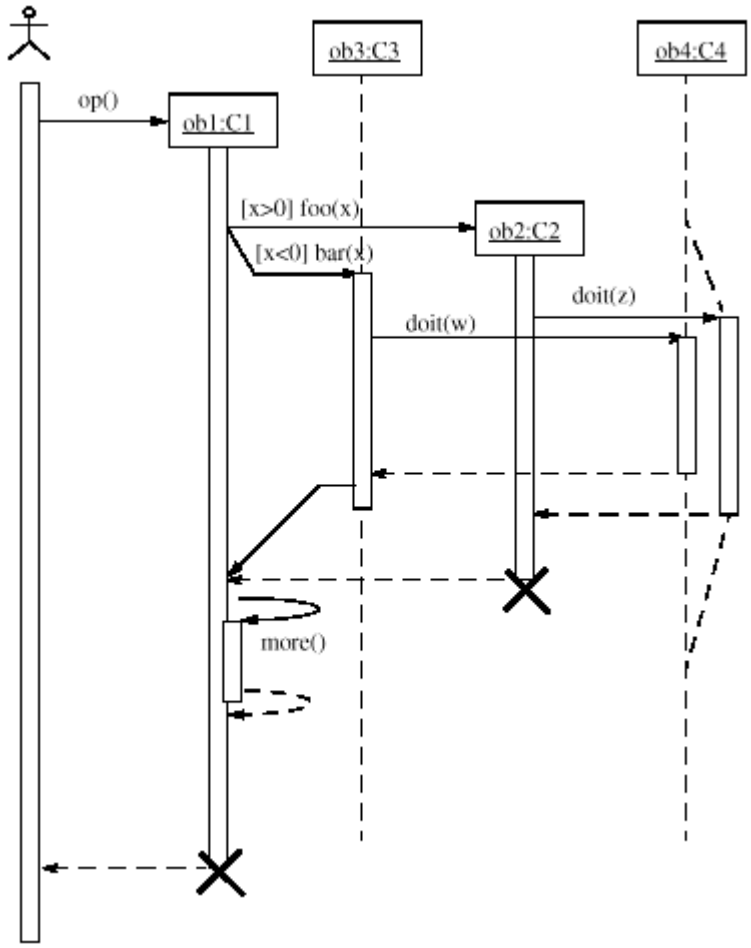
- stimulus asynchrone (*asynchronous stimulus*) ; représenté par une flèche dont la demi-pointe n'est pas remplie (*half stick arrowhead*) :

- retour de procédure (*return from procedure call*) : représenté par une flèche en pointillé dont la pointe n'est pas remplie (*dashed arrow with stick arrowhead*).

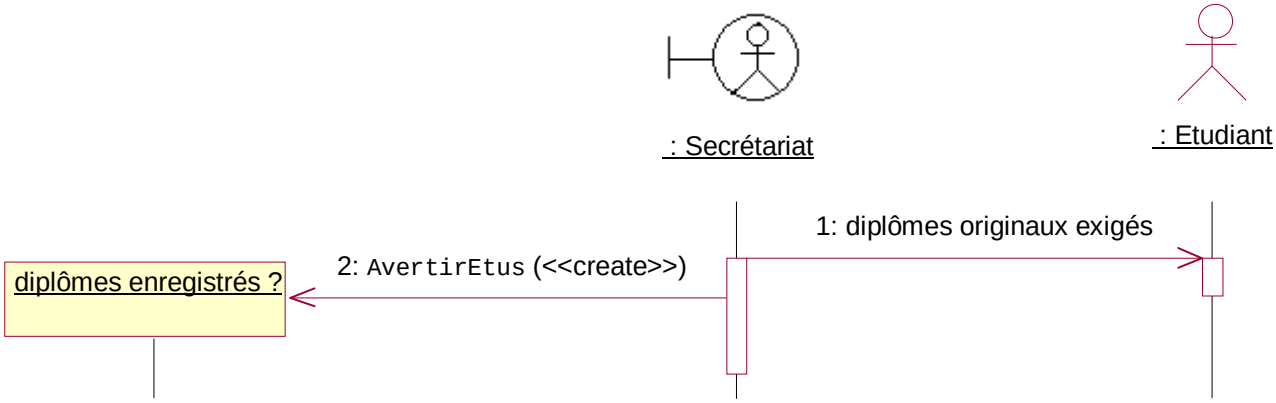
Une interaction (*interaction*) est une séquence de communications i.e. un ensemble partiellement ordonné de messages.
Un **diagramme de séquence** (*sequence diagram*) montre une interaction arrangée en séquence dans le temps (i.e. montre la séquence explicite des stimuli échangés entre les objets).
Un diagramme de séquence a deux dimensions : la dimension verticale représente le temps et la dimension horizontale représente les objets ; les stimuli sont échangés entre les objets.
Une ligne de vie (*lifeline*) montre un objet jouant un rôle spécifique. Un objet peut être créé ou détruit durant la période décrite par le diagramme de séquence.
Une activation (*activation* ou *focus on control*) montre la période durant laquelle un objet est en train d'exécuter une action (directement ou par un sous-programme).
L'exemple ci-dessous présente un diagramme de séquence avec trois objets (`caller`, `exchange`, `receiver`) concurrents (car représentés dans des boîtes avec un cadre épais).

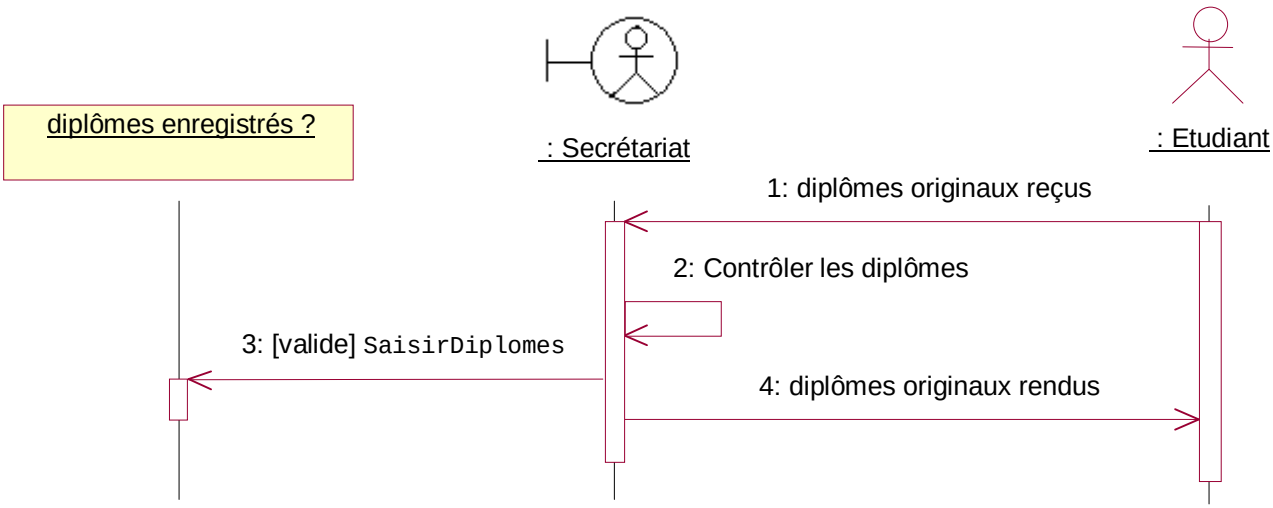


L'exemple ci-après présente un diagramme de séquence avec différentes actions : activation, alternative, récursion, création et suppression. L'acteur déclenche l'opération `op()` qui crée l'objet `ob1`, qui déclenche soit l'opération `foo(x)` si `x>0` en créant l'objet `ob2` soit l'opération `bar(x)` si `x<0` en activant l'objet `ob3`, qui appèle récursivement l'opération `more()` et qui détruit l'objet `ob1` avant de rendre la main à l'acteur. De plus, lorsque l'objet `ob4` termine son action, il rend la main à l'objet qui l'a appelé (`ob2` ou `ob3`).



Les diagrammes de séquence de l'exemple « jouet », pour le scénario `Avertir les étudiants` tout d'abord (ci-dessous) et pour le scénario `Enregistrer les diplômes` ensuite (ci-après) du cas d'utilisation `Gérer les diplômes des étudiants`.




diplômes enregistrés ?
: Secrétariat
: Etudiant
1: diplômes originaux reçus
2: Contrôler les diplômes
3: [valide] SaisirDiplomes
4: diplômes originaux rendus

5.5. Diagramme de collaboration

Un **stimulus** (*stimulus*) est une communication entre deux objets qui s'échangent de l'information dans l'espoir qu'une action (exemples : invocation d'une opération, déclenchement d'un signal, création d'objet, suppression d'objet) s'ensuivra.
Un message (*message*) est la spécification d'un stimulus, précisant les rôles auxquels l'émetteur et le récepteur doivent se conformer.
Les différentes communications ou types de flots de contrôle (*control flow type*) :

- appel de procédure (*procedure call*) ou flot de contrôle imbriqué (*nested flow of control*) : représentés par une flèche dont la pointe est remplie (*filled solid arrowhead*) ;

- flot de contrôle non imbriqué (*flat flow of control*) : représenté par une flèche dont la pointe n'est pas remplie (*stick arrowhead*) ;

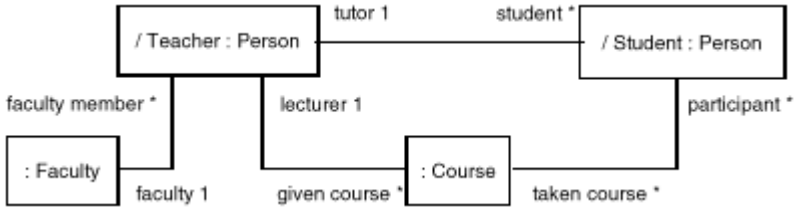
- stimulus asynchrone (*asynchronous stimulus*) ; représenté par une flèche dont la demi-pointe n'est pas remplie (*half stick arrowhead*) :

- retour de procédure (*return from procedure call*) : représenté par une flèche en pointillé dont la pointe n'est pas remplie (*dashed arrow with stick arrowhead*).

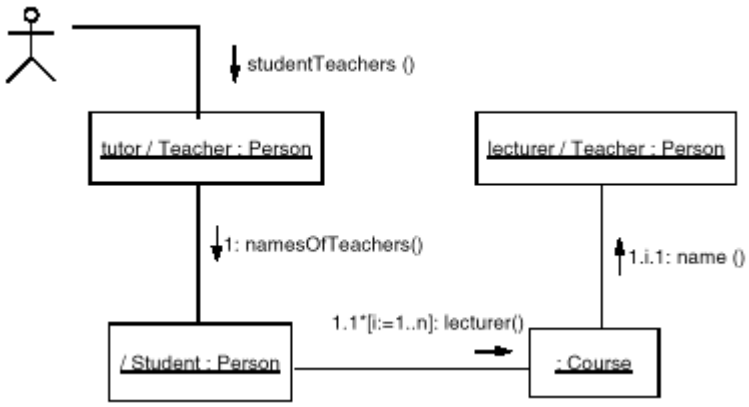
La syntaxe d'une étiquette sur un flot de contrôle : ◁`liste prédécesseurs`▷ `/ [` ◁`garde`▷ `]` ◁`séquence`▷ ◁`valeur retour`▷ `:=` ◁`nom message ou stimulus`▷ ◁`liste arguments`▷ où les prédécesseurs sont représentés par des nombres (séparés par des virgules) et où la séquence est représentée par des termes (séparés par des points) ; un terme a la forme ◁`entier ou nom`▷ ◁`récurrence`▷ où la récurrence peut être `[` ◁`condition`▷ `]` (alternative), `* [` ◁`itération`▷ `]` (répétitive) ou `*||` (traitement parallèle).
Exemples d'étiquettes :

`2: display(x,y)`	message simple
`1.3.1: p:= find(specs)`	appels imbriqués avec une valeur retournée
`[x < 0] 4: inv(x,color)`	message conditionné
`A3,B4/ C3.1*: update()`	synchronisation avec d'autres traitements (`A3` et `B4` doivent être finis) et itération

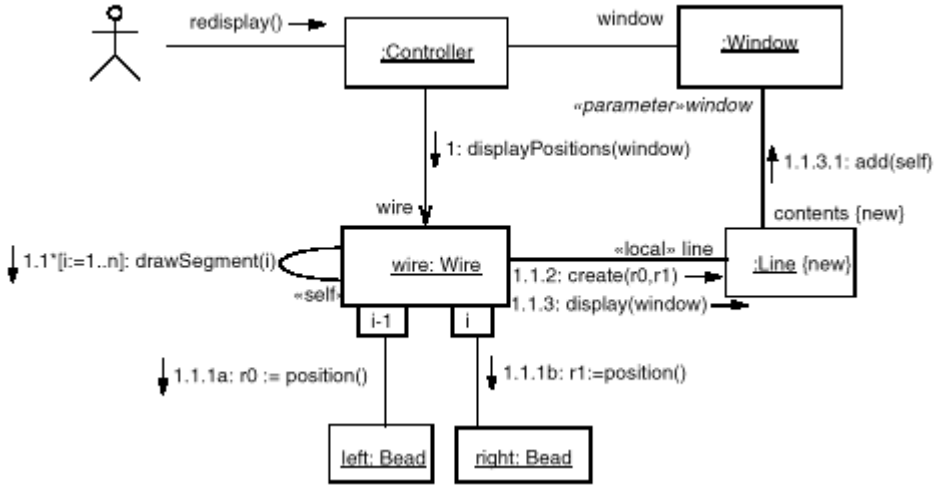
Une collaboration (*collaboration*) est une construction statique qui montre (uniquement) les objets et leurs relations impliqués dans l'accomplissement d'un objectif (ou d'un ensemble d'objectifs liés) inclus dans un système plus général (comportant d'autres objectifs).
Une collaboration permet de décrire la réalisation d'une opération, d'un cas d'utilisation, etc.
Un nom d'objet d'une collaboration est de la forme : ◁`nom objet`▷ `/` ◁`nom rôle`▷ `:` ◁`nom classe`▷ ; par exemple, `O:C` pour un objet nommé `O` de la classe `C` sans préciser de rôle.
Un nom de rôle d'une collaboration est de la forme : `/` ◁`nom rôle`▷ `:` ◁`nom classe`▷ ; par exemple, `/R` pour un rôle nommé `R` sans préciser de classe.
Une interaction (*interaction*) est une séquence de communications i.e. un ensemble partiellement ordonné de messages.
Une interaction est définie dans le contexte d'une collaboration.
Un **diagramme de collaboration** (*collaboration diagram*) montre une collaboration et peut également présenter une interaction (rappel : une collaboration ne montre pas les communications contrairement à une interaction).
Un diagramme de collaboration est un graphe défini soit au niveau des spécifications, soit au niveau des instances. Au niveau des spécifications, un diagramme de collaboration montre les rôles (classes et associations) définis dans une collaboration, et éventuellement les messages. Au niveau des instances, un diagramme de collaboration montre une collection d'objets et de liens, et éventuellement les stimuli … qui de plus doit être conforme au niveau des spécifications.
L'exemple ci-dessous présente un diagramme de collaboration au niveau des spécifications. Les enseignants (des personnes), membres d'une faculté, donnent des cours et à des étudiants (des personnes) ; les étudiants ont un enseignant qui les tuteure.



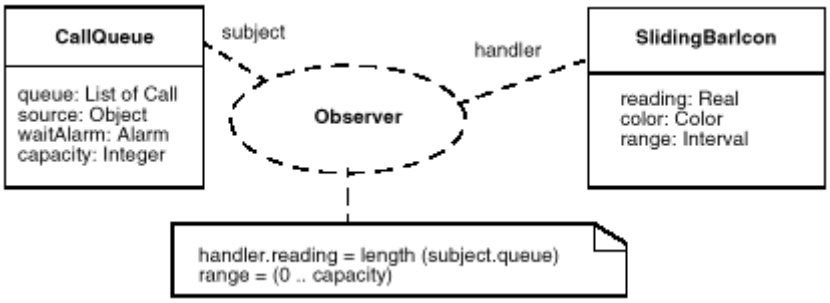
L'exemple ci-après présente un diagramme de collaboration au niveau des instances, avec des objets, des liens et des stimuli. L'acteur demande les enseignants des étudiants (`studentTeachers()`), c'est à dire les noms des enseignants (`namesOfTeachers()`) des étudiants qui suivent (`lecturer()`) des (pour `i=1..n`) cours (`Course`) assurés par des enseignants (le nom `name()` de l'enseignant assurant le `i`ème cours). Plusieurs objets (`tutor` et `lecturer`) jouent ici le même rôle (`Teacher`).



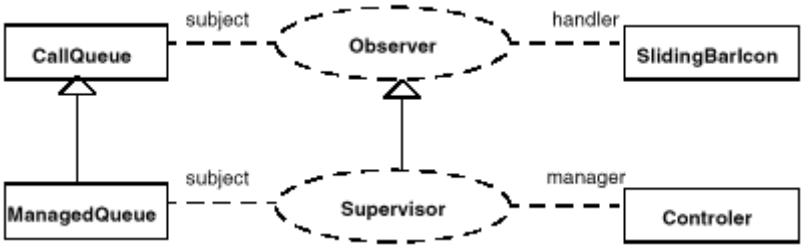
On peut exprimer des contraintes telles que la création d'objets au cours de l'exécution (`{new}`), la suppression d'objets au cours de l'exécution (`{destroyed}`) ou encore la création et la suppression d'objets au cours de l'exécution (`{transient}`).
L'exemple ci-dessous présente un diagramme de collaboration au niveau des instances, avec la création d'objets au cours de l'exécution.



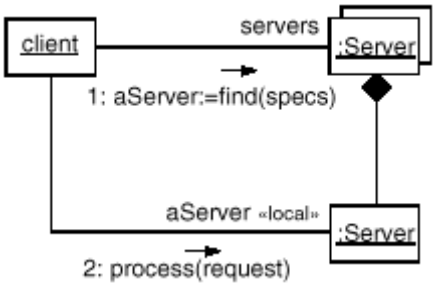
L'exemple ci-dessous montre l'utilisation d'une collaboration (`Observer`) dans un diagramme de classes.



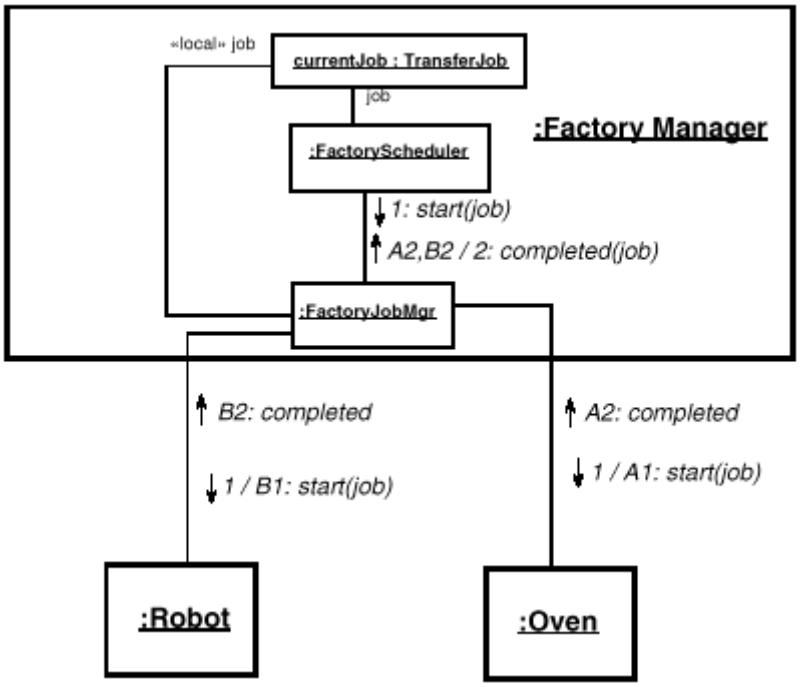
L'exemple ci-dessous illustre la généralisation/spécialisation d'une collaboration (entre `Supervisor` et `Observer`).



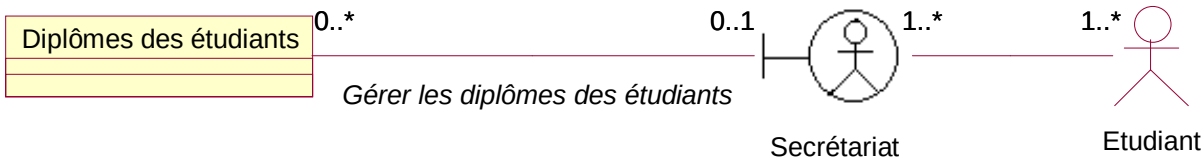
Un multi-objet (*multiobject*) représente un ensemble d'objets. On se sert des multi-objets par exemple lorsqu'une opération s'adresse à l'ensemble entier des objets et non à un seul des objets.
Un multi-objet est représenté par un second rectangle légèrement décalé vers le haut et à droite.
L'exemple ci-après montre un multi-objet (`Server` du haut).

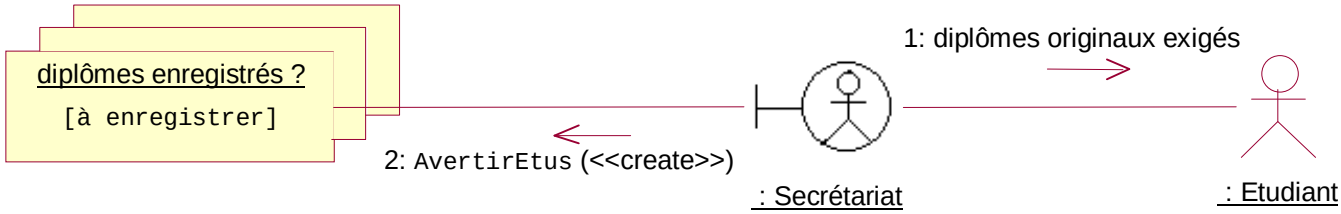


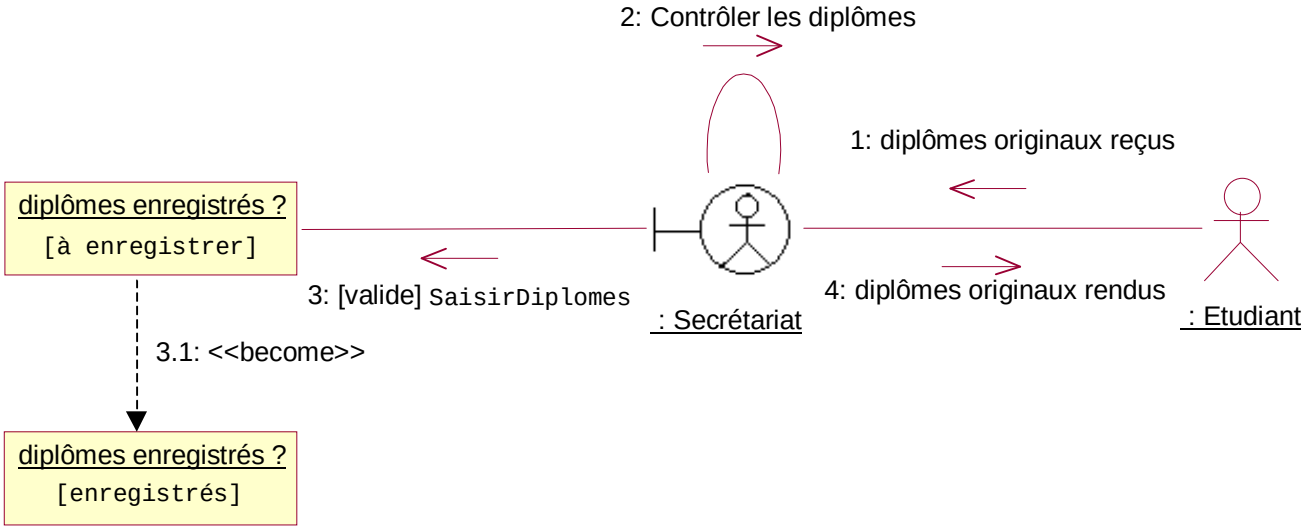
Un objet actif (*active object*) est un objet qui possède un « processus léger » (*thread*) de contrôle et qui peut initialiser l'activité de contrôle. Un objet passif (*passive object*) est un objet qui possède des données mais pas un traitement de contrôle.
L'exemple ci-dessous illustre un objet actif composite. Le gestionnaire de l'usine (*factory manager*) pilote le robot (*robot*) et le four (*oven*).



Les diagrammes de collaboration de l'exemple « jouet » (ci-dessous), au niveau des spécifications (il s'agit en fait d'un diagramme de classes, les classes pouvant être des entités, des travailleurs ou des acteurs) tout d'abord (ci-dessous) et au niveau des instances ensuite pour le scénario `Avertir les étudiants` (ci-dessous) et pour le scénario `Enregistrer les diplômes` (ci-après) du cas d'utilisation `Gérer les diplômes des étudiants`.





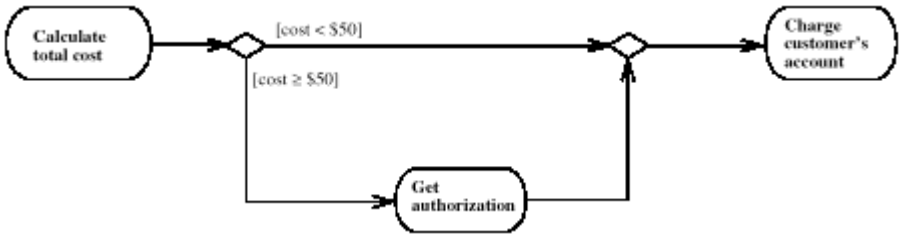

2: Contrôler les diplômes
1: diplômes originaux reçus
diplômes enregistrés ?
[à enregistrer]
3: [valide] SaisirDiplomes
: Secrétariat
4: diplômes originaux rendus
: Etudiant
3.1: <<become>>
diplômes enregistrés ?
[enregistrés]

5.6. Diagramme d'activités

Un événement (*event*) est une occurrence notable.
Un état (*state*) est une situation dans laquelle un objet ou une action satisfait certaines conditions, exécute des actions ou attend des événements.
Une transition (*transition*) est une relation entre deux états qui indique qu'un objet dans l'état source entrera dans l'état cible et exécutera des actions spécifiques quand un événement précis surviendra et si certaines conditions sont satisfaites.
Une **machine à états** est un comportement qui spécifie les suites d'états qu'un objet ou une interaction traverse durant sa vie en réponse à des événements, avec les réponses et les actions.
Une machine à états est un automate fini, déterministe et connexe.
Un **état action** (*action state*) est un état avec une action en entrée et au moins une transition en sortie (qui fait suite à la fin de l'action d'entrée).
Un état action est représenté par un rectangle aux côtés arrondis vers l'extérieur.
L'exemple ci-dessous montre deux états action : celui de gauche est exprimé en langage naturel tandis que celui de droite est exprimé dans un langage de programmation.

Un état action de sous-activité (*subactivity state*) invoque un diagramme d'activité.
L'exemple ci-dessous montre un état action de sous-activité (on le sait grâce au symbole figurant en bas à droite dans l'état action).

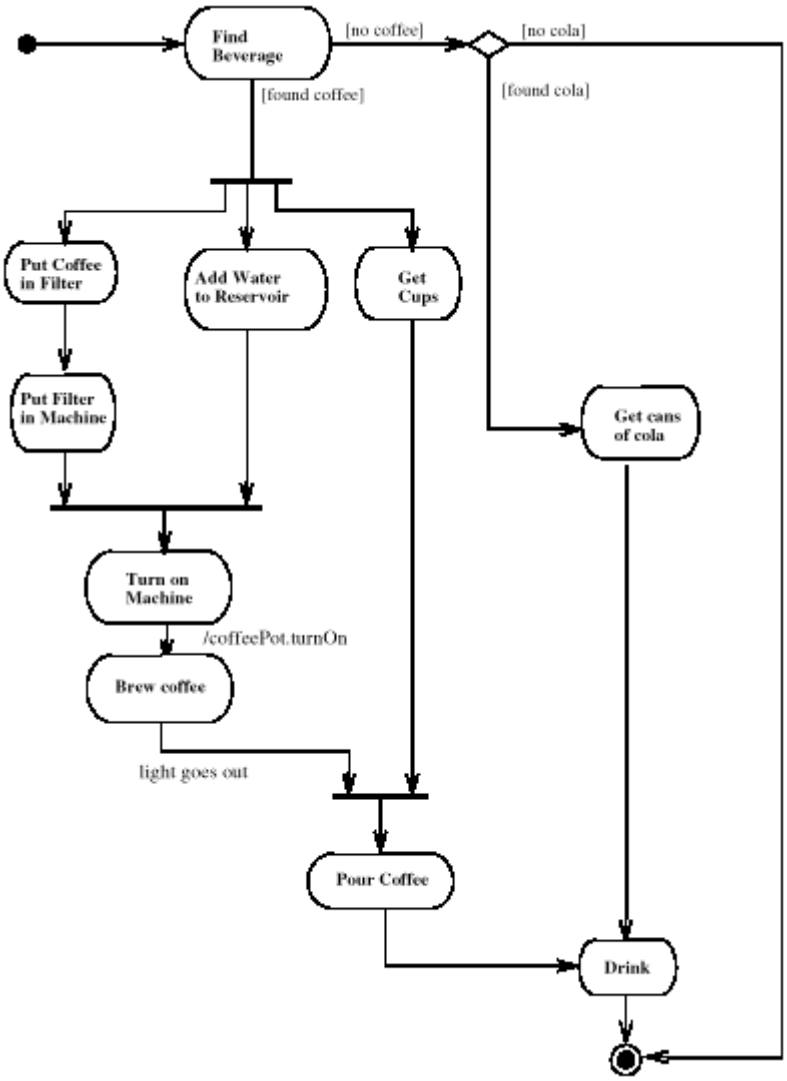
Une **transition automatique** est une relation qui indique le passage d'un état action vers un autre état action, dès la fin du traitement effectué par l'état action source. Il s'agit d'un flot de contrôle.
Une transition est représentée par une flèche.
Une décision (*decision*) permet d'exprimer une condition de garde sur une transition.
L'exemple ci-dessous illustre une décision à prendre (`cost < $50` ou non) suite à l'état action `Calculate total cost` et la fusion vers l'état action `Charge customer's account`.

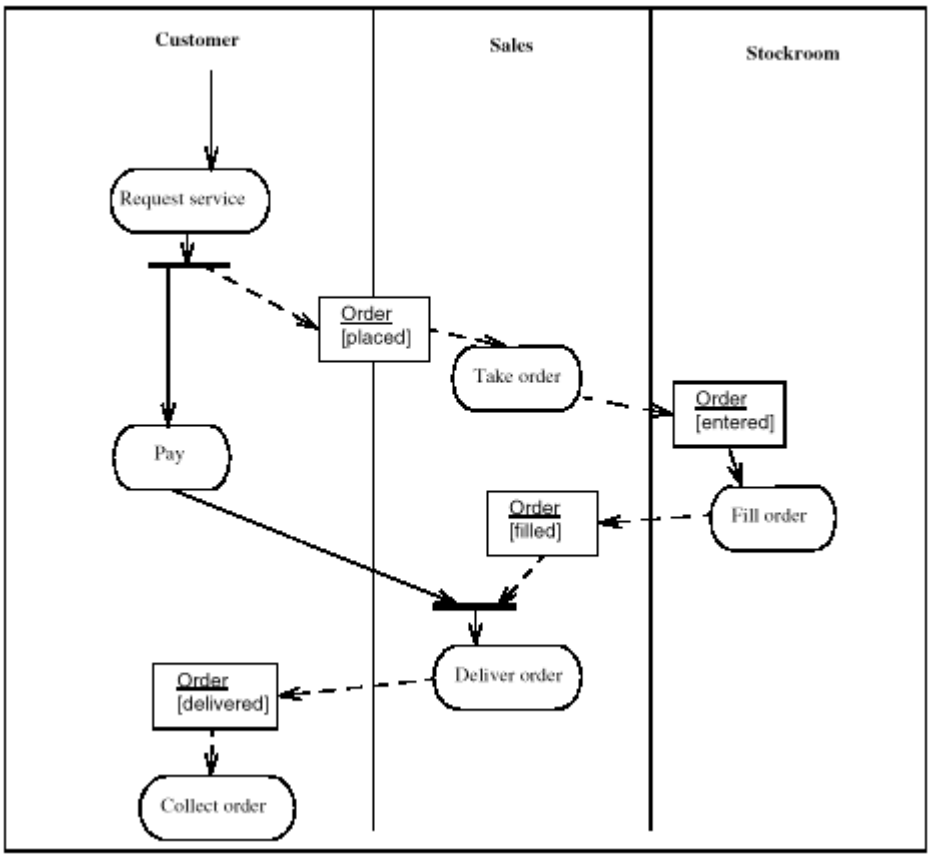


L'algorithme équivalent :

```
Calculate total cost
si cost ≥ $50
alors Get authorization
Charge customer's account
```

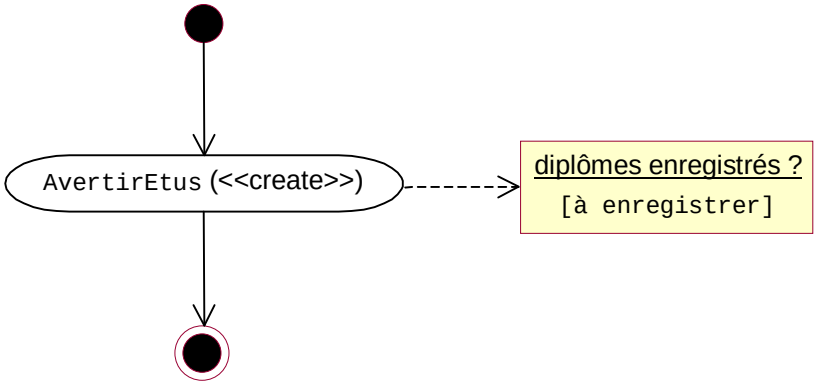
Un **diagramme d'activités** (*activity diagram*) est une variante des machines à états dans laquelle les états correspondent à l'exécution d'actions ou d'activités (i.e. sont des états action) et où les transitions sont automatiques.
Un diagramme d'activités s'utilise pour montrer les événements correspondant à des actions internes (i.e. des flots de contrôle de procédures).
Un diagramme d'activité est attaché dans sa globalité soit à une classe, soit à un cas d'utilisation, soit à un paquetage, soit à l'implémentation d'une opération.
L'exemple ci-après (en haut) illustre un diagramme d'activités pour une personne qui prépare une boisson (l'ordre de ses préférences est le suivant : café, cola, rien). Notez les synchronisations (l'action `Turn on Machine` s'exécute dès que les actions `Put filter in Machine` et `Add Water to Reservoir` sont toutes deux finies) et la parallélisation (une fois l'action `Find Beverage` finie et si la condition de garde (`found coffee`) est vraie, les trois actions `Put Coffee in Filter`, `Add Water to Reservoir` et `Get Cups` peuvent être réalisées en parallèle).
Les couloirs (*swimlanes*) permettent d'organiser les responsabilités des actions et sous-activités selon la classe (le plus souvent, il s'agira des unités organisationnelles).
Il est possible de représenter le flot d'objets (en entrée ou en sortie d'une ou plusieurs actions).
L'exemple ci-après (en bas) illustre un diagramme d'activités montrant les responsabilités (les trois couloirs `Customer`, `Sales` et `Stockroom`) et les flots de l'objet `Order` (dans les états `placed`, `entered`, `filled` et `delivered`) en plus des actions, des flots de contrôle et des synchronisations.

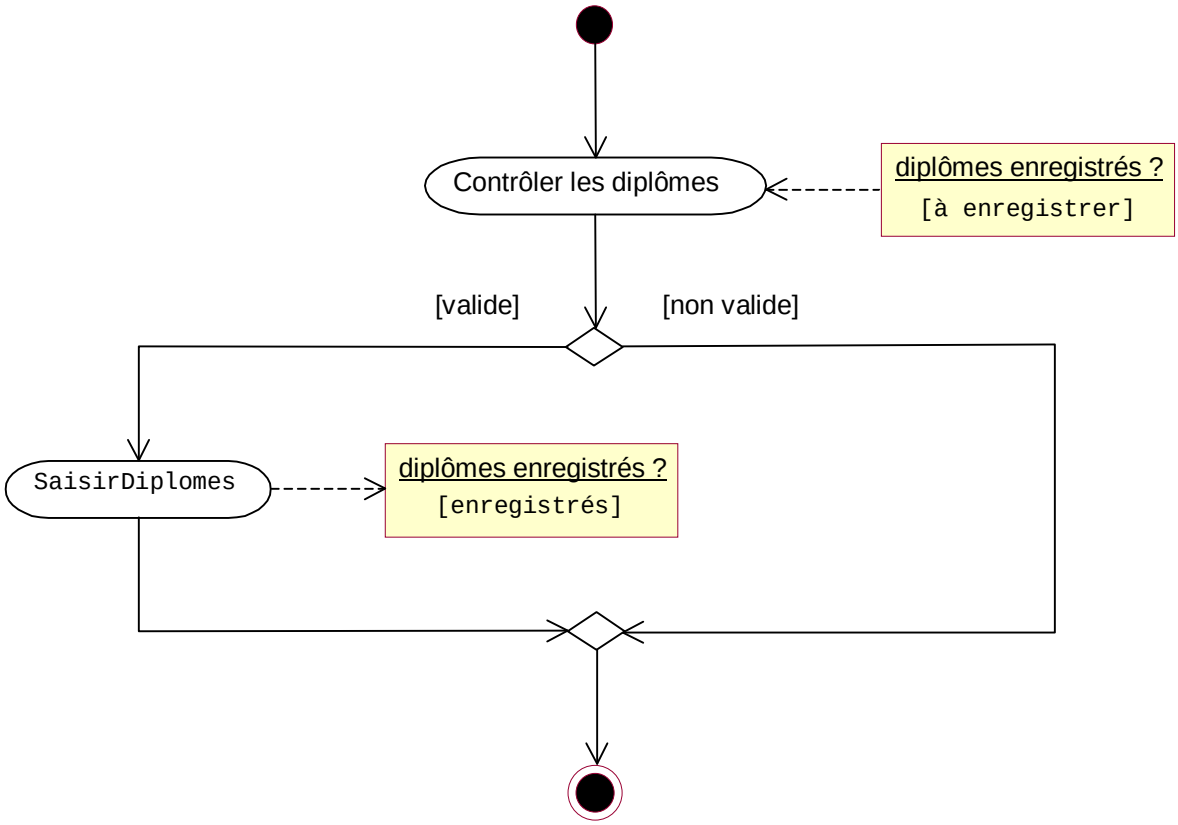

Find Beverage
[no coffee]
[no cola]
[found coffee]
[found cola]
Put Coffee in Filter
Add Water to Reservoir
Get Cups
Put Filter in Machine
Get cans of cola
Turn on Machine
/coffeePot.turnOn
Brew coffee
light goes out
Pour Coffee
Drink


Customer
Sales
Stockroom
Request service
Order [placed]
Take order
Order [entered]
Pay
Order [filled]
Fill order
Deliver order
Order [delivered]
Collect order

On peut préciser graphiquement une action correspondant à l'envoi ou à la réception d'un signal.
Un événement différé (*deferrable event*) est un événement qui n'est pas consommé dès sa survenance mais qui est placé dans une file d'attente (syntaxe : ◁`événement`▷ `/ defer`) pour être utilisé ultérieurement (syntaxe : ◁`événement`▷).
L'exemple ci-après illustre l'envoi (`turnOn`) et la réception (`light goes out`) d'un signal ainsi qu'un événement différé (`light goes out`) qui n'est pas immédiatement traité s'il survient durant les actions `Brew coffee` et `Get cups` (`light goes out / defer`).

Un état action de synchronisation permet de modéliser une machine contenant des régions concurrentes.
L'exécution d'un état action peut être répétée en précisant la multiplicité dans le coin supérieur droit de cet état action. On parle d'invocation dynamique car cette multiplicité peut être évaluée au cours de l'exécution.
Les diagrammes d'activités de l'exemple « jouet » (ci-dessous), pour le scénario `Avertir les étudiants` tout d'abord (ci-dessous) et pour le scénario `Enregistrer les diplômes` ensuite (ci-après) du cas d'utilisation `Gérer les diplômes des étudiants`.




Contrôler les diplômes
diplômes enregistrés ?
[à enregistrer]
[valide]
[non valide]
SaisirDiplomes
diplômes enregistrés ?
[enregistrés]

5.7. Diagramme d'états-transitions

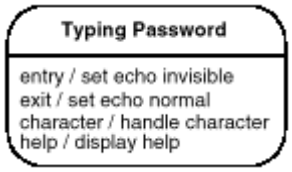
Un **événement** (*event*) est une occurrence notable.
Un événement peut être une condition passant de faux à vrai, la réception d'un signal, la réception d'un appel de procédure, la fin d'une période de temps.
Exemples d'événements faisant référence à une période de temps :

```
after ( 10 seconds since exit from state S )
when ( date = 1/1/2000 )
```

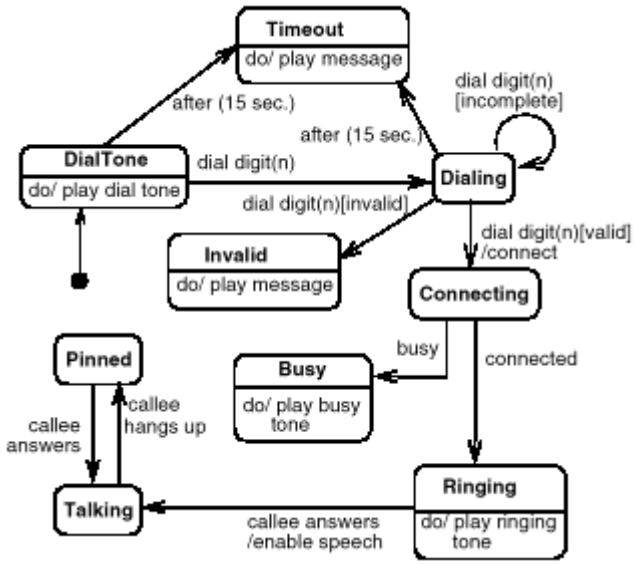
Les événements surviennent un à un.
Un **état** (*state*) est une situation dans laquelle un objet ou une action satisfait certaines conditions, exécute des actions ou attend des événements.
Une **transition** simple (*transition*) est une relation entre deux états qui indique qu'un objet dans l'état source entrera dans l'état cible et exécutera des actions spécifiques quand un événement précis surviendra et si certaines conditions sont satisfaites.
Une transition est représentée par une flèche sur laquelle sont éventuellement mentionnés l'événement, la condition de garde et des actions ou activités.
La syntaxe d'une transition : ◁`événement`▷ `(` ◁`liste paramètres`▷ `) [` ◁`garde`▷ `] /` ◁`suite actions ou activités`▷ où les paramètres (séparés par des virgules) ont la forme ◁`paramètre`▷ `:` ◁`type`▷. Quelques événements sont prédéfinis : `entry` (lors de l'entrée dans l'état), `exit` (lors de la sortie de l'état), `do` (activité exécutée quand l'objet ou l'interaction est dans cet état), `include` (pour identifier l'invocation d'une sous-machine).
L'exemple ci-dessous montre une transition activable sur l'événement d'enfoncement du bouton droit de la souris (`right-mouse-down`), ayant comme paramètre la position (`location`), valide si la position est dans la fenêtre (`location in window`), et générant deux actions pour récupérer l'objet (`object := pick-object (location)`) et le mettre en inversion vidéo (`object.highlight ()`).

```
right-mouse-down (location) [location in window] / object := pick-object (location);
object.highlight ()
```

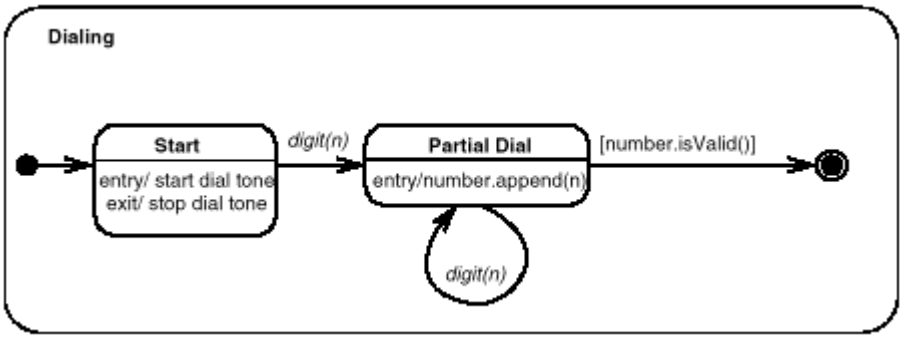
Un état est simple ou composite.
Un état est représenté par un rectangle aux coins arrondis composé d'éventuellement plusieurs compartiments. Les compartiments décrivent le nom de l'état, les transitions internes (i.e. les actions ou les activités internes exécutées lorsque l'objet ou l'interaction est dans cet état), une région graphique (pour un état composite).
L'exemple ci-dessous montre l'état `Typing Password` réagissant à quatre événements (`entry`, `exit`, `character` et `help`).



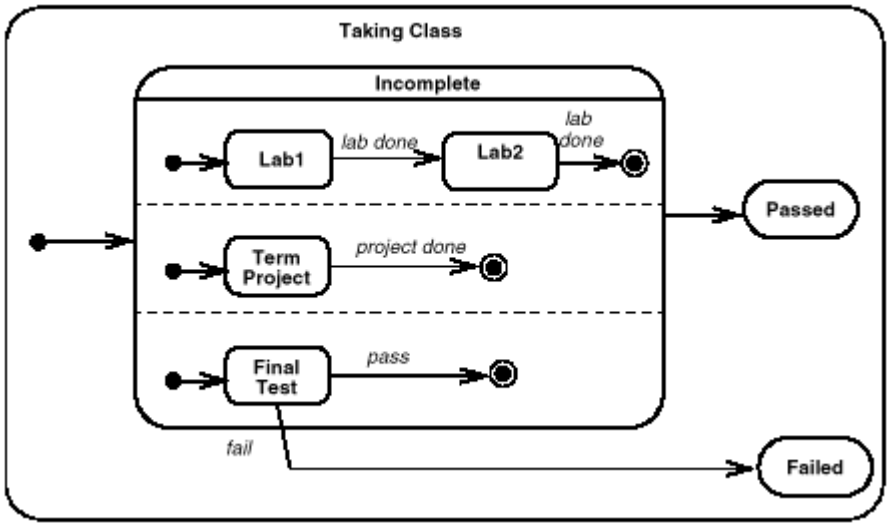
Si un événement ne déclenche pas de transition (de l'état courant vers un autre état, ou interne à l'état courant), il est ignoré et supprimé.
Un **diagramme d'états-transitions** (*statechart diagram*) représente le comportement d'entités (objets, interactions, etc.) susceptibles d'un comportement dynamique en spécifiant ses réponses (états et actions) à la réception des événements intervenant au cours de sa vie. Cela concerne bien évidemment les classes mais aussi les cas d'utilisation, les acteurs, les sous-systèmes, les opérations, les méthodes.
Une machine à états est un automate fini, déterministe et connexe.
Un diagramme d'états-transitions est un graphe qui représente une machine à états (les sommets sont les états ou pseudo-états tandis que les arcs sont les transitions).
L'exemple ci-après illustre un diagramme d'états-transitions d'un appel téléphonique (l'utilisateur vient de décrocher son combiné pour téléphoner). Le pseudo-état initial place l'objet dans l'état `DialTone`, la frappe du premier chiffre (avant 15 secondes) fait passer dans l'état `Dialing`, la frappe d'autres (du deuxième à l'avant-dernier) chiffres (avant 15 secondes) fait boucler dans l'état `Dialing`, la frappe du dernier chiffre fait tenter d'établir la connexion (`/connect`) et fait passer dans l'état `Connecting` si le numéro composé est valide, etc.



Un état composite (*composite state*) est un état divisé en régions (*regions*) concurrentes.
Chaque région doit avoir un pseudo-état initial et un ou plusieurs pseudo-états finaux. Une transition vers l'état composite (super-état) déclenche une transition vers le pseudo-état initial de chaque région. Une transition vers l'un des pseudo-états finaux d'une région correspond à la fin de l'activité de cette région. La fin des activités de toutes les régions correspond à la fin de l'activité du super-état. Une transition depuis l'extérieur de l'état composite vers un sous-état d'une région déclenche toutes les actions d'entrée, à tous les niveaux d'imbrication ; de même, une transition depuis un sous-état d'une région vers l'extérieur de l'état composite déclenche toutes les actions de sortie, à tous les niveaux d'imbrication. Une transition depuis l'état composite équivaut à une transition appliquée à chaque sous-état de la région correspondante (et à tous les niveaux d'imbrication). Un seul événement peut déclencher plusieurs transitions s'il est applicable dans plusieurs régions concurrentes.
L'exemple ci-dessous montre un super-état `Dialing` qui se décompose en deux sous-états séquentiels (`Start` et `Partial Dial`).

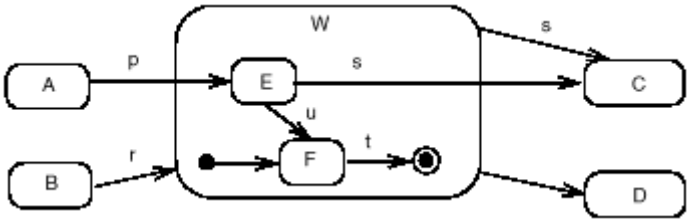


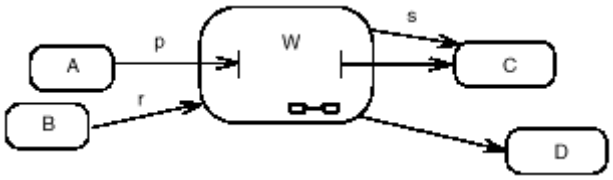
L'exemple ci-après montre un état composite `Incomplete` qui se décompose en trois régions. L'entrée dans le super-état `Taking Class` fait passer les régions dans des états `Lab1`, `Term Project` et `Final Test`. Supposons tout d'abord que l'événement `lab done` survienne : la région du haut passe dans l'état `Lab2`. Supposons ensuite que l'événement `project done` survienne : la région de milieu est donc finie. Si maintenant l'événement `fail` survient, alors les trois régions sont terminées et l'état est `Failed` ; sinon (c'est l'événement `pass` qui est survenu) la région du bas est donc finie et dès que l'événement `project done` surviendra, la région du haut sera finie et l'état sera finalement `Passed`.



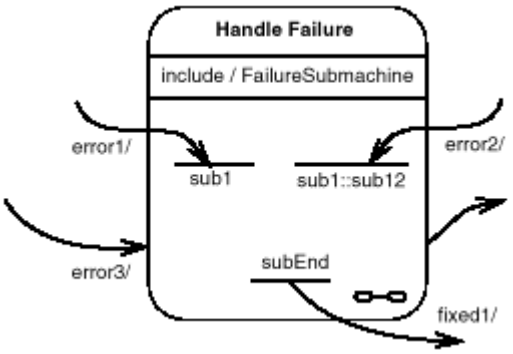
Il est possible de masquer la décomposition d'un état composite.
L'exemple ci-dessous illustre un état composite dont la décomposition est masquée (on le sait grâce au symbole figurant en bas

à droite dans l'état).
Afin d'éviter de dessiner tous les états imbriqués, on peut se restreindre au dessin de l'état le plus général et les transitions subsumées (*subsumed transitions*) peuvent être remplacées par une souche (*stub*).
L'exemple ci-dessous illustre deux diagrammes d'états-transitions, celui de gauche pouvant être abstrait par celui de droite.



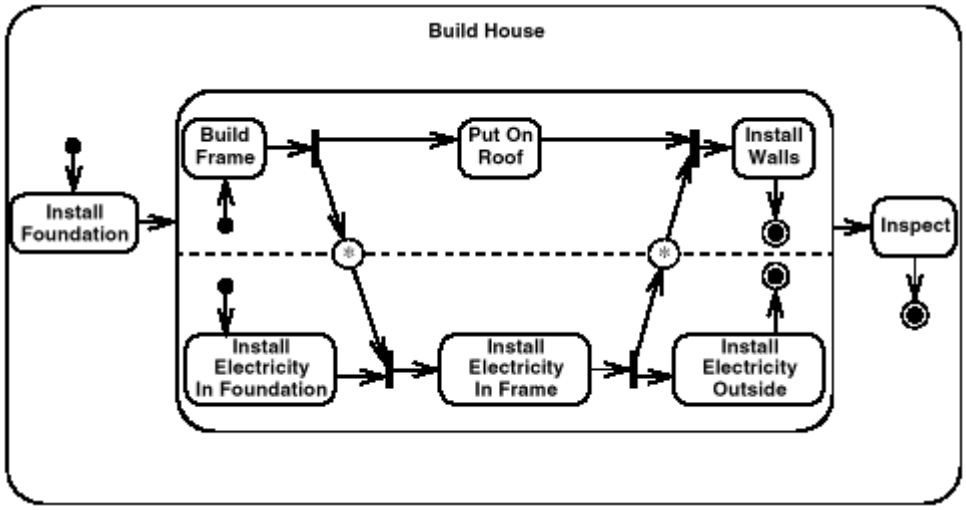


Une sous-machine à états (*submachine state*) représente l'invocation d'une machine à états définie ailleurs. Les transitions en entrée et en sortie peuvent être globales ou concerner n'importe quel sous-état, à tous les niveaux d'imbrication.
L'exemple ci-dessous illustre la sous-machine à états `Handle Failure`. L'événement `error1` fait entrer dans le sous-état `sub1`, l'événement `error2` fait entrer dans le sous-état `sub12` du sous-état `sub1`, l'événement `error3` fait entrer dans le pseudo-sous-état initial de la sous-machine à états, l'événement `fixed` depuis le sous-état `subEnd` fait sortir de la sous-machine à états, et l'arrivée dans l'un des pseudo-sous-états finaux fait sortir de la sous-machine à états.

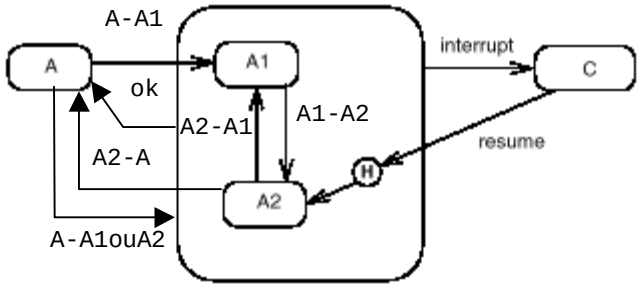


Un état de synchronisation (*synch state*) permet de synchroniser des régions concurrentes. Il est noté par une barre.
L'exemple ci-après illustre plusieurs états de synchronisation. Avant d'installer l'électricité dans la charpente (`Install Electricity In Frame`), il faut que la charpente soit construite (`Build Frame`) et que l'électricité soit installée dans les fondations (`Install Electricity In Foundation`). Par contre, dès que la charpente est construite (`Build Frame`), on peut simultanément poser le toit (`Put On Roof`) et, si l'électricité a été installée dans les fondations (`Install Electricity In Foundation`), installer l'électricité dans la charpente (`Install Electricity In Frame`).

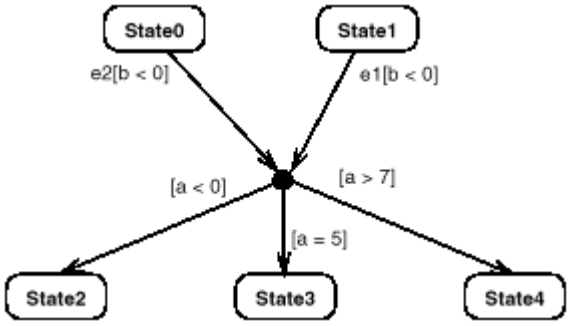
Chaque région peut contenir un indicateur d'état historique (*history state*), noté par la lettre H encerclée, pseudo-état ayant au



moins une transition en entrée (depuis un état extérieur) et au plus une transition en sortie (vers un état de la région).
L'exemple ci-après illustre un diagramme d'états-transitions avec un indicateur d'état historique. On peut imaginer le déroulement suivant : l'objet est dans l'état `A`, franchit la transition `A-A1` et passe dans l'état `A1`, franchit la transition `interrupt` et passe dans l'état `C`, franchit la transition `resume` et passe dans l'état `A2`, franchit la transition `A2-A1` et passe dans l'état `A1`, franchit la transition `ok` et passe dans l'état `A`, franchit la transition `A-A1ouA2` et passe dans l'état `A1`.



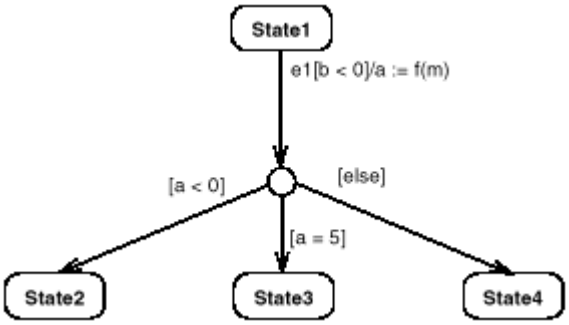
Une transition combinée (*compound transition*) est une succession de traitements (à travers des pseudo-états et des transitions) exécutés comme une transition atomique. Un tel pseudo-état peut être un point de jonction (*junction point*) ou un point de choix dynamique (*dynamic choice point*), commun à un ou plusieurs états en entrée comme en sortie.
L'exemple ci-dessous illustre six transitions combinées : deux transitions se rejoignent (*merge*) et se séparent (*split*) en trois transitions. Le point de jonction est commun aux deux transitions provenant des états `State0` et `State1`. Les trois transitions gardées sortant du point de jonction représentent un point de branchement statique (*static branch point*).
L'évaluation des gardes de sortie d'un point de jonction se fait avant d'emprunter la transition y entrant (ainsi, on ne peut pas rester dans un tel pseudo-état).



L'algorithme équivalent à la transition combinée allant de l'état `State0` vers l'état `State2` :

> `si` l'objet est dans l'état `State0` `et` l'événement `e2` survient `et` `b<0` `et` `a<0`
> `alors` l'objet passe dans l'état `State2`
> `sinon` l'objet reste dans l'état `State0`

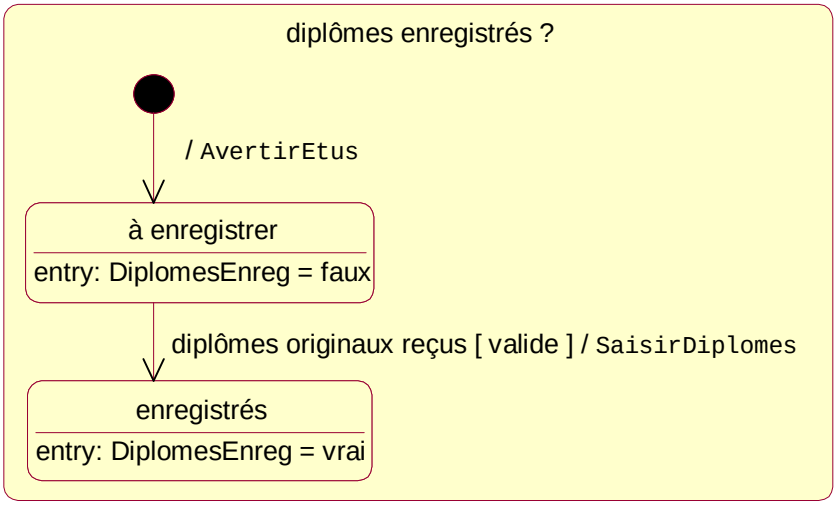
L'exemple ci-après illustre trois transitions combinées : une transition se sépare (*split*) en trois transitions. Les trois transitions gardées sortent du point de choix dynamique. L'évaluation des gardes de sortie d'un point de choix dynamique se fait en y entrant (ainsi, on pourrait rester dans un tel pseudo-état mais ce cas doit être rendu impossible, grâce aux conditions sur les gardes des transitions en sortie qui doivent imposer le passage d'une et d'une seule transition en sortie).



L'algorithme équivalent :

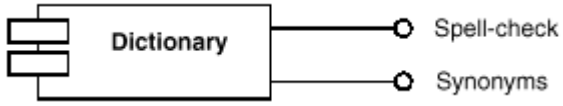
si l'objet est dans l'état `State1` et l'événement `e1` survient et `b<0`
alors `a := f(m)`
l'objet passe dans le pseudo-état de choix dynamique
selon que
`a<0` : l'objet passe dans l'état `State2`
`a=5` : l'objet passe dans l'état `State3`
sinon l'objet passe dans l'état `State4`
sinon l'objet reste dans l'état `State1`

Le diagramme d'états-transitions de l'exemple « jouet » (ci-dessous), indiquant si les diplômes (pour chaque étudiant) ont été enregistrés.

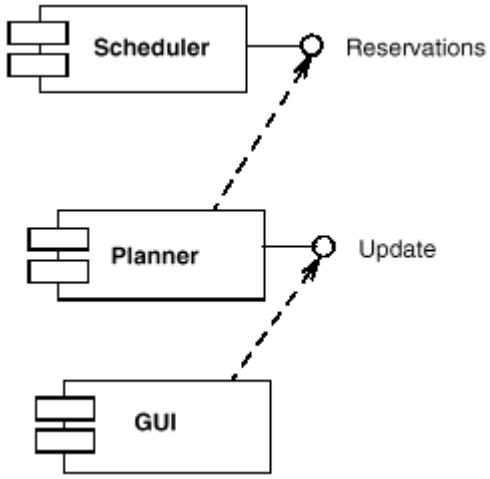


5.8. Diagramme de composants

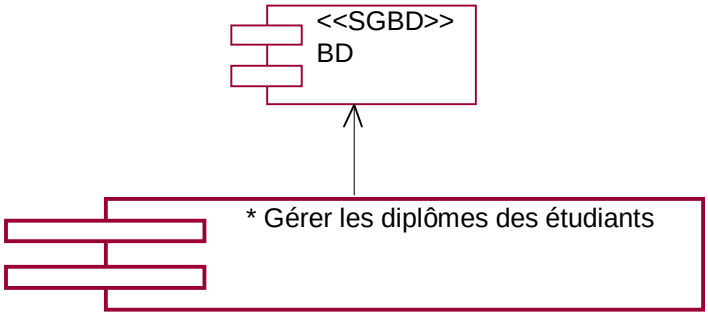
Un **composant** (*component*) représente une partie distribuable d'un système.
Au niveau métier, dans un système humain, les composants comprennent les règles de gestion et les documents ; au niveau informatique, les composants contiennent les programmes (sources, exécutables).
L'exemple ci-dessous représente (au niveau type) le composant `Dictionary`. De plus, il propose deux interfaces.



Un **diagramme de composants** (*component diagram*) montre les dépendances entre les composants (entre les informations dans un système humain, ou pour le compilateur).
Un diagramme de composants est représenté par un graphe de composants connectés par des relations de dépendance (et éventuellement par des relations de composition). Un composant ne peut être représenté qu'au niveau type (et non niveau des instances).
L'exemple ci-dessous illustre un diagramme de composants ayant trois composants (`Scheduler`, `Planner` et `GUI`) et deux relations de dépendance (entre le composant `Planner` et l'interface `Reservations` du composant `Scheduler`, et entre le composant `GUI` et l'interface `Update` du composant `Planner`).

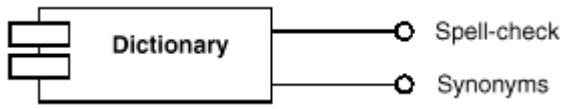


Le diagramme de composants de l'exemple « jouet » (ci-dessous).

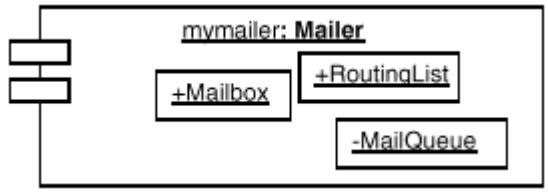


5.9. Diagramme de déploiement

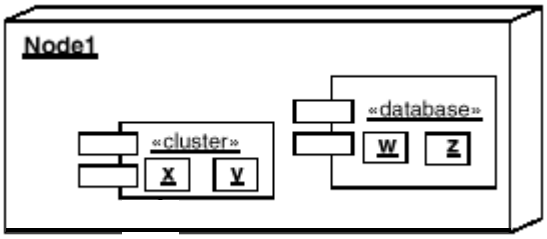
Un **composant** (*component*) représente une partie distribuable d'un système. Il peut contenir des programmes (sources, exécutables) ou, pour un système humain, des documents de gestion notamment.
Un composant peut être représenté soit au niveau type, soit au niveau des instances.
L'exemple ci-dessous représente le composant `Dictionary` au niveau type. De plus, il propose deux interfaces.



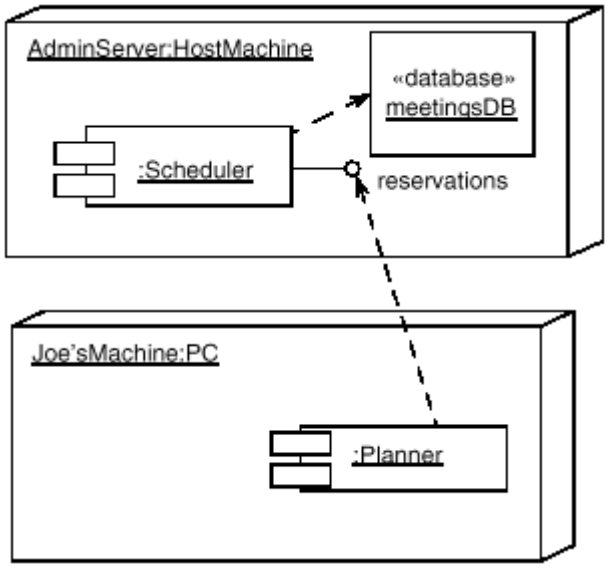
L'exemple ci-dessous représente le composant `mymailer` au niveau des instances. De plus, il possède trois objets en cours d'exécution.



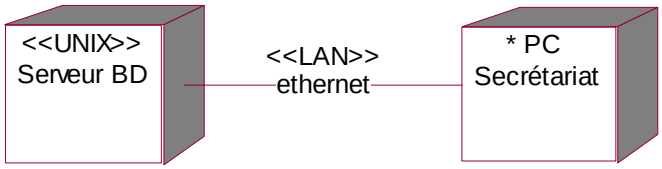
Un **nœud** (*node*) est un objet physique correspondant à une ressource de traitement (mémoire et processeur). Il s'agit d'ordinateurs, de ressources humaines, de traitements mécaniques.
Un nœud peut être représenté soit au niveau type, soit au niveau des instances.
L'exemple ci-dessous représente (au niveau des instances) le nœud `Node1`. De plus, il possède deux composants en cours d'exécution.



Un **diagramme de déploiement** (*deployment diagram*) montre la configuration des organes de traitements (*run-time processing element*) ainsi que les composants logiciels, les traitements et les objets qui y vivent.
Au niveau métier, ces organes de traitements correspondent aux travailleurs et aux unités organisationnelles tandis que les composants logiciels comprennent les procédures et les documents utilisés par ces travailleurs et ces unités organisationnelles.
Un diagramme de déploiement est représenté par un graphe de nœuds connectés par des associations de communication. Un nœud peut posséder des instances de composants ; un composant peut posséder des objets.
L'exemple ci-dessous illustre un diagramme de déploiement ayant deux nœuds (`AdminServer:HostMachine` et `Joe'sMachine:PC`), une relation de dépendance (entre le composant `:Planner` et l'interface `reservations` du composant `:Scheduler`) et un objet (`meetingsDB`).



Le diagramme de déploiement de l'exemple « jouet » (ci-après).


<<UNIX>>
Serveur BD
<<LAN>>
ethernet
* PC
Secrétariat