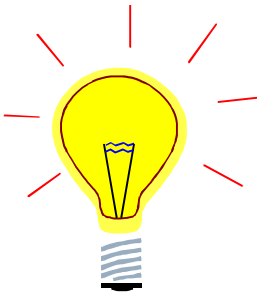


UML

(Unified Modeling Language)

Table des matières

Introduction

Introduction

Le Logiciel

Etat des lieux:
Programme de plus en plus complexe, donc temps de conception et de développement de plus en plus long.

- Le test des logiciels est encore mal réglé ("bug"!!!).
- La maintenance est une part grandissante de l'informatique.
- L'évolution technique produit une complexité croissante.
- La demande de logiciel est de plus en plus faite pour l'utilisateur grâce à des interfaces conviviales (I.H.M.). Cela demande de plus en plus de ressources.

La complexité croissante entraîne une susceptibilité à l'effondrement accru.

- Ex. d'application dépassant les capacités de l'homme :
  - ◊ Système de gestion du monde réel : Contrôle du trafic routier
  - ◊ Système de simulation : Réseaux de neurones.
- Ex. de système complexe : Un ordinateur personnel.
  - ◊ C'est : une unité centrale, un écran, un clavier, ...
    - ∗ Une unité centrale c'est : un CPU, de la mémoire, un bus
      - ÷ Un CPU c'est : un ALU, des registres, un bus, ...
- Nature hiérarchique d'un système complexe.
- Collaboration des parties pour un bon fonctionnement.
- indépendance des parties.

Système complexe

Les 5 règles qui permettent de déterminer qu'un système est complexe.

- Le système complexe est souvent composé comme une hiérarchie de sous-systèmes reliés entre eux, et ainsi de suite, jusqu'à atteindre le niveau le plus bas.
- Le choix des sous-systèmes primaires est arbitraire et dépend en grande partie de l'observateur du système.
- Les liaisons "intra composants" sont généralement plus fortes que les liaisons "inter composants". Les premières sont des dynamiques haute fréquence des composants (celles qui concernent la structure interne des composants). Les secondes sont des dynamiques basse fréquence (celles qui concernent l'interaction entre composants).
- Les systèmes hiérarchiques sont généralement composés d'un petit nombre de types de sous-systèmes.
- Un système qui fonctionne est toujours issu d'un système plus simple, qui a lui aussi fonctionné.
  Un système complexe conçu à partir de rien ne fonctionne jamais et ne peut pas fonctionner de façon satisfaisante, même en le "bidouillant". Il faut tout refaire en recommençant à partir d'un système plus simple qui fonctionne.

Méthodes de conception

- Principe de base : la décomposition
  - Un projet logiciel demande la décomposition du problème en sous-système.
- Approche structurée
  - Guidée par les traitements : module principal, décomposition progressive. On obtient des modules fonctionnels (C, Fortran).
- Approche par les données
  - Analyse en structure de données (Cobol).
- Approche par les connaissances
  - Réservée au développement de systèmes experts. Adaptée aux cas où le domaine demande une forte expertise.
- Approche objet
  - Conception d'unités autonomes qui encapsulent les données et les traitements sur ces données.

Ex. construire une fenêtre graphique :

- ◊ Approche structurée :
  - dessiner le cadre
  - le remplir
  - dessiner le bandeau (qui est un cadre)
  - etc.
- ◊ Approche objet :
  - demander à l'instance rectangle *cadre* de se dessiner
  - lui demander de se remplir
  - demander à l'instance rectangle *bandeau* de se dessiner
  - etc.

Objet : Rappel et définition

- Un objet est une combinaison entre données et fonctions.
- L'association des 2 est supérieure à la somme des parties.
- Dans une programmation objet tout devrait être objet.
- Les objets sont inertes, seuls des messages qui leur sont adressés permettent de modifier leur état interne.
- "L'objet" se définit succinctement avec 3 termes :
  - ◊ La classe
  - ◊ La hiérarchie de classes
  - ◊ L'instance

♦ La classe est un "moule de fabrication".
Ce n'est qu'un **type** qui déclare au compilateur comment est constituée la structure **et non une occupation mémoire** (variable).

Elle permet d'avoir un regroupement dans une seule entité des données, appelées *attributs* , et des fonctions appelées *méthodes*. Ce "regroupement" s'appelle *l'encapsulation*.

♦ La hiérarchie de classes est une organisation.

Elle permet de *récupérer* ce qui a déjà été conçu, et de rajouter à une classe existante des propriétés , attributs et/ou méthodes , afin d'obtenir une autre classe plus spécifique.

♦ L'instance joue pour la classe le même rôle que la variable pour la structure de données.

C'est elle qui va occuper de la place en mémoire, à qui l'on va envoyer des messages, et qui va alors modifier les états de ses attributs.

nota : On parle d'attributs et de méthodes de classe et d'instance, mais cela ne recouvre pas les mêmes choses.

Dans les classes :

- Les *attributs* sont des **déclarations** de variables que l'on désire utiliser. Ces variables sont nommées.
- Les *méthodes* sont des **déclarations** de codes que l'on désire implémenter. Le codage lui-même est souvent fait à ce moment de déclaration.

Dans les instances :

- Les attributs d'instances sont les **espaces mémoires** qui ont été réservés par le compilateur, et dont les adresses correspondent à leurs noms dans la classe.
- Les *méthodes* sont des **appels** de fonctions que l'on veut faire exécuter. ***Ce sont ces appels de fonctions et leurs paramètres , qui sont des messages***.

Un programme objet est constitué de :

- Une série de classes ou de hiérarchies de classes.
- Une série d'instances.
- Un séquencement de messages sur les instances.

Utilisation des objets

L'abstraction

- L'abstraction consiste à prendre les aspects essentiels d'un objet sans s'occuper des aspects secondaires.
- L'abstraction c'est *se focaliser sur ce qu'est un objet et ce qu'il fait*, plutôt que se demander *comment* il est fait.
- L'héritage[1] et le polymorphisme[2] renforcent la possibilité d'abstraction.

L'encapsulation

- Cela permet de séparer les *aspects extérieurs* qui seront *publics* pour tous les utilisateurs de l'objet, *des aspects internes* qui n'ont à priori *pas à être connus*.
- Les aspects extérieurs sont les *dialogues* que l'on peut faire avec l'objet, ce sont en général des méthodes, et donc ce sont des *messages*.
- L'implémentation d'un objet doit pouvoir être modifiée sans que les applications qui l'utilisent n'aient de conséquences trop importantes.
- L'encapsulation permet aux éléments d'un programme de ne pas devenir trop interdépendants. La moindre petite modification doit pouvoir se faire sans trop de "casse".

---

1 Le principe fondamental de l'héritage est le "look up". Si le message envoyé à l'objet n'existe pas dans la classe même , il faut aller regarder dans les classes "hiérarchiquement" supérieures à celle-ci, jusqu'à trouver la méthode. Si l'on ne trouve pas de méthode correspondant au message, le compilateur doit générer une erreur.

2 Le polymorphisme est la capacité d'un langage objet à avoir le même nom de méthodes dans différentes classes. Ces méthodes étant dans des classes différentes, peuvent avoir des codes différents.
Ex. On veut créer une hiérarchie de classes d'objets graphiques. Après analyse on a les classes : *Point, Droite, Polygones*, etc. Ces figures seront affichées, respectivement effacées, avec des méthodes appelées toutes *Show*, respectivement *Hide*. Il est évident que les codes de ces méthodes sont différents, mais leurs noms sont identiques. Cela permet d'utiliser naturellement les instances et de leur envoyer des messages portant le même nom.

Conception par objets

- La démarche est plus itérative que descendante.

- Il y a 5 étapes
  - Identifier les entités du domaine
  - Analyser les propriétés de ces entités et leurs relations afin de structurer le domaine.
  - Détecter les opérations que savent faire ces entités.
  - Décrire ces opérations qui seront les messages.
  - Décrire le lancement du programme.

Identifier les entités du domaine

- Se focaliser sur ce qui doit être manipulé, c'est le principe de réification
  - Définition

    La réification c'est représenté informatiquement sous la forme d'un objet quelque chose de réel.
  - Principe

    Si l'on parle d'une chose en lui attribuant des propriétés, ou si cette chose doit être manipulée, alors il faut la réifier.

- Il faut obtenir de "bonnes classes", pour cela :
  - Prendre les classes évidentes.
  - Donner une liste exhaustive des entités du domaine.
  - Analyser les informations et les classer.
  - Faire parler le spécialiste du domaine, car souvent :

    Nom = classes

    Verbe = méthodes

Analyser les propriétés

- Organiser et relier les classes
- Une propriété est une valeur simple.
  - nombres, booléens , chaîne de caractères , ...
- Une relation est un lien entre objets plus complexes.
  - Le lien qui unit le conducteur à sa voiture :
    est-conducteur-de , est-conduit-par
- Une propriété peut se ramener à un lien.

Détecter les opérations

- Déterminer les messages auxquels doivent répondre les objets.
  - Pour un objet figure géométrique :
    S'afficher, s'éteindre, se déplacer, ...
- Déterminer le comportement de manière la plus abstraite possible.
  - Définir un comportement par défaut.

Décrire ces opérations

- Phase de codage

Décrire le lancement

- Démarrer en instanciant quelques classes initiales.
- Le déroulement du programme sera une suite de messages.

Erreur

Attention à ne pas :

- Faire des tests répétés.
- Confondre héritage et composition.
  héritage = "**est un**"
    une *droite* est un point de *départ* et a un point *d'arrivée*.
  composition = "**a un**"
    un *polygone* est un point de *départ* et a des droites qui sont ses *côtés*.
- Manipuler des objets au loin.
  Le point de la droite du polygone de ... ; mieux vaut avoir un point et lui affecter celui issu de la droite du ...
- Penser en terme de fonction.
  Classique pour ceux qui "viennent" de la programmation structurée : Fortran, C, Pascal , ...
- Penser en terme de données.
  Classique pour ceux qui "viennent" de la programmation par les données : Cobol, ...
- Ne pas bien définir les messages des objets.
  Trop de messages complexifie inutilement, pas assez ne permet pas de manipuler correctement.
  Juste ceux qu'il faut pour avoir tout par *combinaison* de messages.

Analyse

Analyse

- Il faut décrire de manière *précise*, *concise* , *correcte* et *compréhensible* un ***modèle du monde réel***.
- Etapes :
  - ◊ L'analyse commence par un cahier des charges qui peut être incomplet. Il faut le rendre le plus complet possible et mettre en évidence ses ambiguïtés et incohérences.
  - ◊ Le système réel décrit dans le cahier des charges devra être compris, et ses caractéristiques les plus importantes abstraites dans un modèle.
  - ◊ Le modèle d'analyse comprend les 2 aspects des objets :
    - – la structure statique (modèle objet).
    - – le séquencement et les interactions (modèle dynamique).
- Un problème réel peut avoir un modèle plus important que l'autre.

  Ex. Un problème comportant des contraintes de temps comme les interfaces utilisateurs , a un modèle dynamique important. Les problèmes comportant beaucoup de calculs comme les compilateurs ont un modèle statique plus important.
- L'analyse ***doit être un processus itératif***. Une première esquisse de modèle est faite, puis enrichie jusqu'à ce que le problème entier soit compris.
- ***L'analyste doit communiquer avec le client*** afin de clarifier les "points obscurs".

Modélisation des objets

- Etapes nécessaires à la construction d'un modèle objet :
  - identification des objets et des classes
  - préparation d'un dictionnaire des données
  - identification des associations entre objets
  - identification des attributs et des liens
  - organisation et simplification des classes d'objet utilisant l'héritage
  - vérification que tous les chemins nécessaires existent
  - itérations afin d'affiner le modèle
  - groupement des classes en module

Identification des classes d'objets

- Enumérez les classes d'objets pouvant exister à partir de la description écrite du problème.

  Ex. "Un système de réservation pour la vente de tickets pour les pièces jouées dans divers théâtres" les classes pourront être : *Réservation , Système , Tickets , Pièces* et *Théâtre*.

- Ne vous préoccupez pas d'héritage ou de classe de haut niveau.

  Ex. "Un système d'enregistrement de livres dans une bibliothèque"
  Les classes pourront être : *Livres, Magazines, Journaux, Cassettes audio* et *vidéo*. On pourra les organiser ultérieurement en catégories. *Livres, Périodiques, Cassettes*

- Evitez les détails d'implémentation informatique. Toutes les classes ne sont pas explicites dans la description.

  Ex. Oubliez les listes chaînées, les tableaux et autres sous-programmes. Toutes les classes doivent avoir un sens dans le domaine de l'application.
  Physique : Maisons, Employés, etc.
  ou concept : Trajectoire , Zones de stationnement , etc.

Sélection des classes pertinentes

- Eliminez les classes qui ne sont pas nécessaires.
  - *Les classes redondantes*
    Si 2 classes expriment la même information, celle dont le nom est le plus significatif doit être gardée.
    Ex. Si *client* décrit bien une personne qui embarque sur un vol , *passager* est plus significatif.
  - *Les classes non pertinentes*
    Si une classe n'a rien à voir avec le problème. Ce n'est pas toujours évident.
    Ex. Si *vendeur de tickets de théâtre* n'est pas très utile, *personnel de théâtre* doit l'être.
  - *Les classes vagues*
    Une classe doit être clairement décrite.
    Ex. Si *enregistrement de transaction* est vague, *transaction* ne l'est pas.
  - *Les attributs*
    Certains noms décrivant une classe sont en fait des noms d'attributs.
    Ex. nom , âge , poids.

Seule l'importance de l'indépendance d'une propriété peut en faire une classe.

– *Les opérations*

Si un nom décrit une opération qui s'applique à des objets et n'est pas manipulée elle-même, alors ce n'est pas une classe.

Ex. si nous sommes un fabricant de téléphones, alors *appels téléphoniques* est une séquence d'opérations qui doit être dans le modèle dynamique et pas dans une classe d'objets.

– *Les rôles*

Le nom d'une classe doit refléter sa nature et non le rôle qu'elle doit jouer dans une association.

Ex. *Propriétaire* serait de peu d'intérêt pour un constructeur de voiture. *Personne* ou *client* serait plus juste (achat à crédit, location, etc.).

– *Les constructions d'implémentation*

Les constructions étrangères au système réel doivent être éliminées du modèle d'analyse.

Ex. *Listes, Chaînes , Tableaux ,* sont presque toujours des détails d'implémentations.

Préparation d'un dictionnaire des données

- Les mots isolés ont de multiples interprétations. Préparer un dictionnaire pour toutes les entités à modéliser.

Banque :	Institut financier qui gère les *comptes* des *clients*.
Compte :	Compte unique dans une *banque*, sur lequel les *transactions* sont appliquées. Les comptes peuvent être de diverses sortes.
Client :	Détenteur d'un ou plusieurs *comptes* dans une *banque*. Un client peut être : une personne, ou une entreprise. La même personne possédant plusieurs comptes dans plusieurs banques est considérée comme plusieurs clients.
Transaction :	Requête complète unique pour réaliser des opérations sur un *compte*.

Identification des associations entre objets

- Toute dépendance entre 2 ou plusieurs classes est une association.
- Les associations correspondent généralement à des expressions verbales.
  Personne et *Société* sont en relation par l'association *Travaille-pour*.
- Les associations peuvent être implémentées de diverses manières, mais ces implémentations doivent être ignorées dans le cadre du modèle d'analyse, afin de garder une marge de manoeuvre dans la conception.

Sélection des bonnes associations

- Eliminez les associations incorrectes :
  - *Associations entre classes supprimées*
    Si une classe est supprimée, alors l'association doit être éliminée ou redéfinie dans d'autres classes.
  - *Associations non pertinentes*
    Eliminez toute association qui sort du contexte du problème.
    "Le système gère les accès concurrents". La concurrence est inhérente aux objets du monde réel. C'est l'algorithme d'accès qui doit gérer les concurrences.
  - *Actions*
    Une association doit décrire une propriété structurelle et non un événement transitoire.
    "Une banque accepte les cartes de crédit". Décrit une partie de l'interaction entre le client et la banque , et non une relation permanente.
  - *Associations ternaires*
    Presque toutes les associations impliquant 3 classes ou plus peuvent être décomposées en associations binaires.
    "La société verse des salaires à des personnes". Peut être reformulé par : "*La société emploie des personnes*" avec des valeurs salaires pour chaque lien *Société/Personne*.
    On peut avoir une association ternaire *"Le professeur donne cours dans une salle"* ne peut pas être formulé par des associations binaires.

- *Associations dérivées*

  Omettez les associations qui peuvent être formulées à l'aide d'autres associations car elles sont redondantes.

  "Grand-parent de". Peut être reformulé à l'aide d'une paire d'associations *"parent de"*

- *Associations mal nommées*

  Ne dites pas comment ou pourquoi une situation se produit, décrivez-la.

  "Les ordinateurs de la banque entretiennent les comptes" est la description d'une action reformulée par *"La banque possède des comptes"*

- *Nom de rôle*

  Ajoutez des noms de rôle quand cela se justifie. Le nom de rôle décrit le rôle que joue une classe dans l'association du point de vue de l'autre classe.

  Dans l'association *"Travaille-pour"*, *Société* a le rôle *Employeur* et *Personne* a le rôle *Employé*.

- *Associations qualifiées*

  Un qualificatif distingue les objets sur l'extrémité "plusieurs" de l'association.

  "L'entreprise a été fondée dans la ville de". *Nom de l'entreprise* et *Ville* identifient *Entreprise* de manière unique.

- *Multiplicité*

  Spécifiez les multiplicités mais sans y mettre trop d'effort car elles évoluent avec la phase d'analyse.

  "Un chef dirige plusieurs ouvriers". Ecarte la possibilité d'une gestion matricielle ou d'un employé ayant diverses responsabilités.

- *Associations manquantes*

  Ajoutez les associations manquantes que vous découvrez.

Identification des attributs

- Les attributs sont des propriétés d'un objet.
- Les attributs ne doivent pas être des objets.
- Les attributs correspondent à des noms suivis d'expressions de possession. Les adjectifs correspondent à des valeurs particulières prises par les attributs.
  Ex. La couleur rouge de la voiture. La position haute du curseur.
- Retenez les attributs les plus importants d'abord. Les détails plus fins viendront ensuite.

Sélection des attributs

- Eliminez les attributs incorrects :
  - *Objets*
    Si c'est l'existence indépendante d'une entité qui est importante, plutôt que ses valeurs, alors c'est un objet.
    Pour une *adresse*, *Ville* est un attribut alors que pour un *recensement* , *Ville* serait un objet.
  - *Qualificatifs*
    Si la valeur d'un attribut dépend d'un contexte particulier, alors essayez de le transformer en qualificatif.
    n° d'employé n'est pas la caractéristique unique d'une personne , il qualifie *l'association La société emploie la personne*.
  - *Identificateurs*
    Les identificateurs permettent de référencer un objet. Ils ne doivent pas être cités dans le modèle.
  - *Attributs de liens*
    Si une propriété dépend de la présence d'un lien, alors elle est attribut de lien et pas d'objet.
  - *Valeurs internes*
    Si un attribut décrit la valeur interne d'un objet invisible depuis l'extérieur d'un objet, alors éliminez-le de l'analyse.

- *Détails fins*
  Ne pas mettre les attributs qui n'affecteront pas la plupart des opérations.
- *Attributs discordants*
  Un attribut qui semblerait différent et sans relation avec les autres attributs peut caractériser une classe qui devrait être scindée en 2 classes distinctes.

Affinage en utilisant l'héritage

- L'héritage peut être ajouté
  - en généralisant les aspects communs à plusieurs classes dans une super-classe (de bas en haut).
  - en affinant les classes existantes dans des sous-classes (de haut en bas).
- L'héritage multiple peut être utilisé, mais seulement si absolument nécessaire car il augmente la complexité.
- Les attributs et les associations doivent être affectés au plus haut niveau possible de la hiérarchie.

Vérification des chemins d'accès

- Suivez les chemins afin de vérifier s'ils conduisent à des résultats significatifs. Y a-t-il des questions sans réponses ? si oui elles indiquent des manques d'informations.

Itérations de la modélisation

- Il est impossible d'avoir un modèle correct en une seule passe. Il faut faire une suite d'itérations.
- Certains raffinements ne peuvent être faits qu'après les analyses dynamique et fonctionnelle.

◊ Indices qui permettent de déceler des objets manquants :

- *Assymétries dans les associations et les généralisations*
  Ajoutez des classes par analogie.
- *Attributs disparates*
  Scindez la classe de façon que chaque partie soit cohérente.
- *Difficulté de généraliser correctement*
  Une des classes peut avoir 2 rôles. Scindez-la , une des 2 parties doit être cohérente.
- *Opération sur une classe cible incorrecte*
  Ajoutez la classe cible manquante.
- *Duplication d'associations ayant le même nom et le même but*
  Généralisez en créant la super-classe manquante.
- *Un rôle façonnant nettement la sémantique d'une classe*
  Il faut en faire une classe à part entière.

◊ Indices de classes inutiles :

- *Défaut d'attributs , d'opérations et d'associations sur une classe*
  A quoi sert-elle?

◊ Indices d'associations manquantes :

– *Chemin d'accès manquant pour les opérations*
Ajoutez une association de sorte que les requêtes trouvent une réponse.

◊ Indices d'associations inutiles :

– *Informations redondantes dans les associations*
Supprimez les associations qui n'apportent pas d'informations supplémentaires.
– *Défaut d'opérations au travers d'une association*
Si aucune opération n'utilise un chemin donné , l'information n'est certainement pas nécessaire.

◊ Indices de mauvais placement d'une association :

– *Noms des rôles ayant un sens trop large ou trop fin*
Déplacez l'association vers le haut ou le bas dans la hiérarchie.

◊ Indices de mauvais placement d'un attribut :

– *Nécessité d'accéder à un objet par l'intermédiaire de l'un de ses attributs*
Essayez une association qualifiée.

Groupement des classes en modules

- Il est important de grouper les classes en feuillets et modules pour des besoins de dessin et de visualisation.
- Les classes très liées seront regoupées
  Ex. Le modèle d'un système d'exploitation peut comprendre des modules pour le contrôle de processus, la gestion de périphériques, des fichiers et de la mémoire.
- La taille des modules est variable.

Modélisation dynamique

- Commencez l'analyse en vous intéressant aux événements.
- Résumez les séquences d'événements permis pour chaque objet à l'aide d'un diagramme d'états.
- Etapes :
  - ◊ Préparation des scénarios pour des séquences typiques.
  - ◊ Identification des événements entre objets.
  - ◊ Préparation d'un suivi d'événements pour chaque scénario.
  - ◊ Construction d'un diagramme d'états.
  - ◊ Vérification de la correspondance des événements entre objets pour vérifier l'homogénéité de l'ensemble.

Préparation des scénarios

- Préparez un dialogue typique entre l'utilisateur et le système afin de donner une idée du comportement du système.
- Ces scénarios montrent les interactions importantes, les formats d'affichage, l'échange d'information.
- Préparez en premier les scénarios sans condition d'erreur. Ensuite on prend en considération les cas exceptionnels et les erreurs humaines.
- Pour chaque événement identifiez l'acteur qui le provoque.
- Le format des messages échangés n'affecte pas la logique des informations échangées.

Format d'interface

- Presque toutes les interactions peuvent être découpées en 2 parties :
  - La logique de l'application
    Elle peut être simulée par des procédures factices
  - L'interface utilisateur
    Il est difficile d'évaluer une interface utilisateur sans la tester.

Identification des événements

- Examinez les scénarios pour identifier les événements externes.
- Les événements inclus : chaque signal, entrée, décision, interruption, transition et action depuis ou vers l'utilisateur.
- L'action de transmission d'information par un objet est un événement.
  Ex. *"Rentrez le mot de passe"* est un événement de l'objet externe *Utilisateur* vers l'objet de l'application *Gestion des cartes bleues*.
  Certains des flots d'informations sont implicites : *"Déliver les billets"* est un événement de *Gestion des cartes bleues* vers *Utilisateur*.
- Vous devez décider si les différences de valeurs sont suffisamment importantes pour les distinguer.
  Ex. *"Les entrées au clavier"* sont généralement considérées comme un même événement. Néanmoins taper sur *"valider"* peut être un événement distinct dans la mesure où l'application réagit différemment.
- Affectez les événements aux classes d'objets qui les émettent et les reçoivent.
- Représentez chaque scénario sous forme d'un suivi d'événements.

Construction d'un diagramme d'états

- Préparez un diagramme d'états pour chaque objet dont le comportement dynamique n'est pas trivial, en indiquant les événements que l'objet reçoit et ceux qu'il émet.
- Chaque scénario ou suivi d'événements correspond à un chemin dans le diagramme d'états.
- L'intervalle entre 2 événements est un état. Donnez un nom à chaque état.
- Repérez les boucles dans le diagramme. Remplacez les séquences finies d'événements par des boucles.
- Dans une boucle le premier et le dernier état sont identiques.
  Ex. *Insérer 2 fois 1 franc* dans un distributeur automatique revient à y insérer *2 francs*.
- Vous aurez fini votre diagramme d'états lorsqu'il prendra en compte tous les scénarios et qu'il traitera tous les événements pouvant survenir sur un objet dans chacun de ses états.
  Ex. Pour un exemple de diagramme d'états voir page 52.

Vérification de la correspondance des événements entre les objets

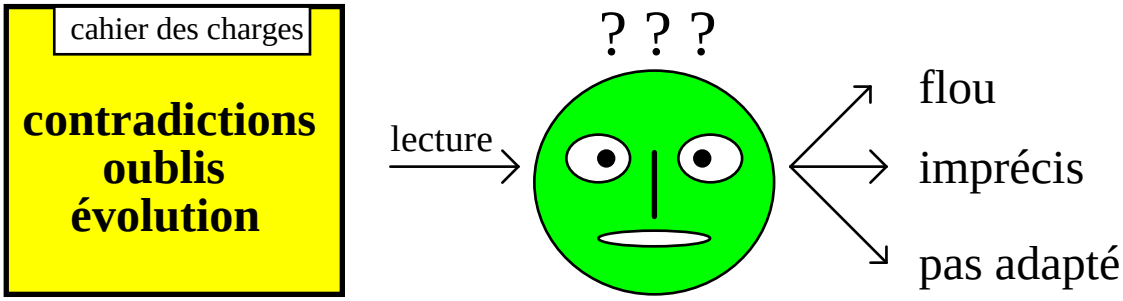
- Vérifiez la complétude et la cohérence quand les diagrammes d'états sont terminés.
- Chaque événement doit lier un émetteur et un récepteur.
- Les états sans suivant ni précédent sont très suspects.
- Prenez garde aux erreurs de synchronisation lorsque des événements arrivent à des instants aléatoires.

Les cas d'utilisation
(USES CASES)

Les cas d'utilisation (uses cases)

"Meilleur des cas classiques"

- Les besoins sont souvent exprimés comme une suite de :
  - le système devra faire ...
  - le système fera éventuellement ...
  - il faut absolument ...
  - il serait intéressant de ...
- En conséquence le cahier des charges est flou et entraîne :
  - des contradictions
  - des oublis
  - des imprécisions
  - une évolution constante du cahier des charges
- Cela implique une analyse qui part sur de mauvaises bases.



Les cas d'utilisation

- Ils recentrent l'expression des besoins sur les utilisateurs.
- Cette partition de l'ensemble des besoins simplifie la complexité de la détermination des besoins.
- La conception du système, donc son utilisation, est bien faite pour les utilisateurs car elle a été pensée pour satisfaire les besoins réels.
- Formaliser les besoins est simple : on le fait dans le langage des utilisateurs, ce qui permet de les valider.
- Cela oblige les utilisateurs à structurer leurs demandes.

Modèle de cas d'utilisation

Le modèle comprend :

- Les acteurs
- Le système
- les cas d'utilisation eux-mêmes

Les acteurs

Ils représentent un rôle joué par une entité. Les candidats les plus courants sont : les utilisateurs, les clients, les vendeurs, les fournisseurs, les autres systèmes, etc. Toutes les choses et les personnes extérieures à un système qui interagissent avec lui.

- Il existe 4 grandes catégories d'acteurs :
  - ♦ Les acteurs principaux
    Ce sont les personnes qui utilisent les fonctions principales du système.
  - ♦ Les acteurs secondaires
    Ont y regroupe les personnes qui effectuent des tâches administratives ou de maintenance.
  - ♦ Le matériel externe
    Ce sont les dispositifs matériels incontournables qui font partie de l'application.
  - ♦ Les autres systèmes
    Ce sont les systèmes avec lesquels l'application doit interagir.

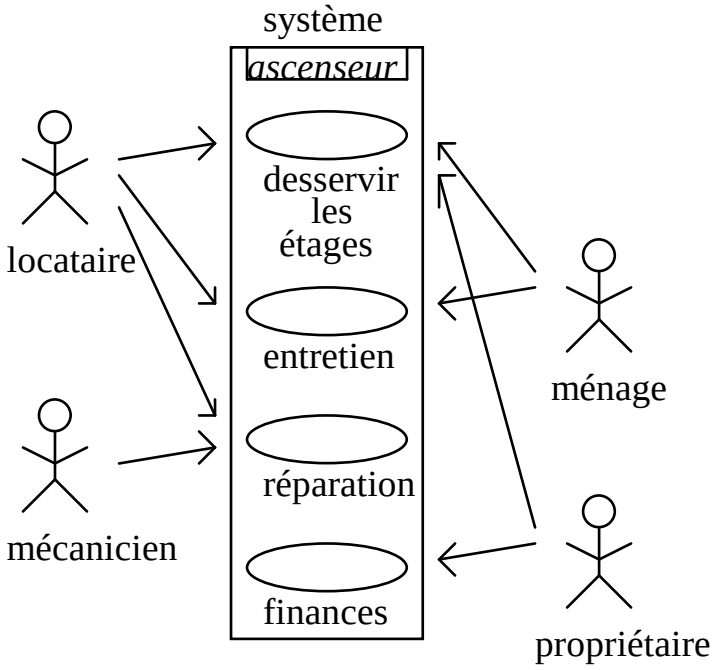
Ex.

Dans un modèle de distribution de billet, les acteurs principaux sont les clients, les acteurs secondaires les personnes qui rechargent les caisses, le matériel externe peut être l'imprimante, enfin l'autre système peut être le groupement bancaire qui gère les parcs de distributeurs.

Représentation

Chacun des acteurs est figuré par un petit personnage qui déclenche des cas d'utilisation. Ces derniers sont représentés par des ellipses contenues dans le système dessiné sous forme d'un rectangle que l'on pourra omettre.

Ex. Un ascenseur d'immeuble.



Les cas d'utilisation

- Les cas d'utilisation se trouvent en précisant, acteur par acteur, les scénarios du point de vue de l'utilisateur.
- Les **cas d'utilisation** doivent être vus comme des **classes** dont les **instances** sont les **scénarios**.

Relation entre cas d'utilisation

Il y a 3 types de relations entre les cas d'utilisation.

- la relation de communication
  La participation de l'acteur est signalée par une flèche entre l'acteur et le cas d'utilisation. Le sens de la flèche indique l'initiateur de l'interaction.

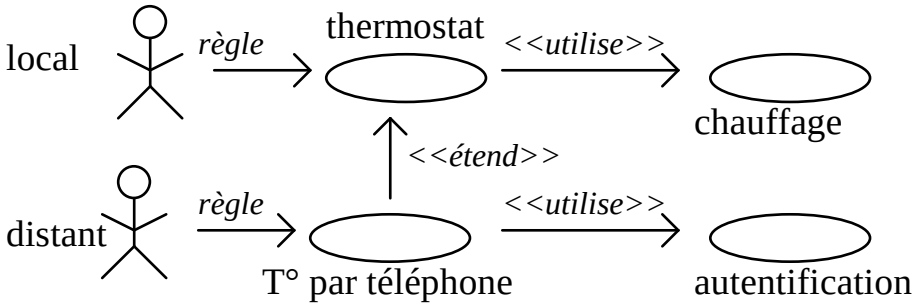
ex.

- la relation d'utilisation
  Une instance du cas d'utilisation source comprend aussi le comportement décrit par le cas d'utilisation destination.

ex.

- la relation d'extension
  Le cas d'utilisation source étend le comportement du cas d'utilisation destination.

ex.



Construction des cas d'utilisation

Un cas d'utilisation procure à l'utilisateur un résultat. Il doit être simple et décrit clairement. Le nombre d'acteurs qui interagissent avec le cas d'utilisation est très limité.

Questions à se poser lors de la construction des cas d'utilisation :

- quelles sont les tâches de l'acteur ?
- quelles informations l'acteur doit-il créer, sauvegarder ... ?
- l'acteur devra-t-il informer le système des changements externes ?
- le système devra-t-il informer l'acteur des conditions internes ?

La description des cas d'utilisation comprend :

- le début du cas d'utilisation
  L'événement qui déclenche le cas d'utilisation.
  Il doit être clairement dit : "le cas d'utilisation débute quand *k* se produit".
- la fin du cas d'utilisation
  L'événement qui cause l'arrêt du cas d'utilisation.
  Il doit être clairement dit : "quand *p* se produit le cas d'utilisation s'arrête".
- l'interaction entre le cas d'utilisation et les acteurs
  Décrit clairement ce qui est dans le système et ce qui est au dehors.
- les échanges d'informations
  Ce sont les paramètres des interactions entre le système et les acteurs.

- la chronologie et l'origine d'informations
  Qui décrivent quand le système a besoin d'information et quand il les enregistre.
- les répétitions de comportement
  Qui peuvent être décrites au moyen de pseudo code.

Ex. tant que une condition
    Faire qqchose (qui modifie la condition)
    Fin tant

- les situations optionnelles
  Elles doivent être représentées de manière uniforme dans tous les cas d'utilisation.

Ex. l'acteur choisit un des éléments suivants :
    Choix 1 choix 2 choix 3
    Puis l'acteur continue à faire ...

La recherche de cas d'utilisation est difficile car il faut trouver le bon niveau d'abstraction (la quantité de détails qui doivent apparaître). Il faut aussi faire la distinction entre un cas d'utilisation et un scénario.

Il n'y a pas de réponses toutes faites mais celles que l'on peut apporter aux 2 questions suivantes facilitent le travail :

- peut-on faire une activité indépendamment des autre ?
  2 activités qui s'enchaînent toujours font probablement partie du même cas d'utilisation.
- peut-on regrouper certaines activités pour les tester, les documenter, ou les modifier ?
  Si oui, ces activités font sûrement partie du même cas d'utilisation.

Nota : Lorsqu'un cas d'utilisation devient trop complexe (10 pages maximum) il est possible de le découper en cas d'utilisation plus petit. Cependant cette opération sera faite quand tout le cas aura été décrit.

Elaboration des cas d'utilisations

L'élaboration des cas d'utilisations se fait en groupe ; à plusieurs on a plus d'idées que tout seul.

Les cas sont identifiés grossièrement et décrits en une phrase ou 2.

Une fois ce travail réalisé, des groupes issus de l'équipe approfondissent chaque cas.

♦L'aide de critères simples permet de distinguer un bon cas :

* existe-t-il une brève description du cas ?
* bien cerner les conditions de démarrage et d'arrêt du cas.
* l'interaction entre l'acteur et le cas est-elle satisfaisante ?
* existe-t-il des étapes communes avec un autre cas ?
* les termes utilisés ont-ils bien la bonne signification ?
* toutes les alternatives ont-elles été prises en compte ?

♦Pièges à éviter :

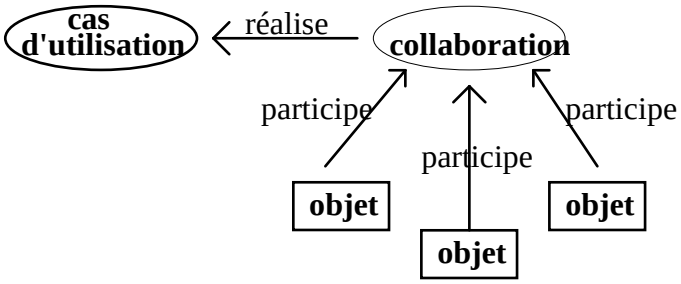
♣est-ce que tous les besoins ont été alloués à un cas ?
♣est-ce qu'un même besoin a été alloué à plusieurs cas ?
♣existe-t-il des cas non référencés par les besoins ?

nota : Dans un système, un trop grand nombre de cas indique un manque d'abstraction. Un système moyen comporte environ de 10 à 20 cas.

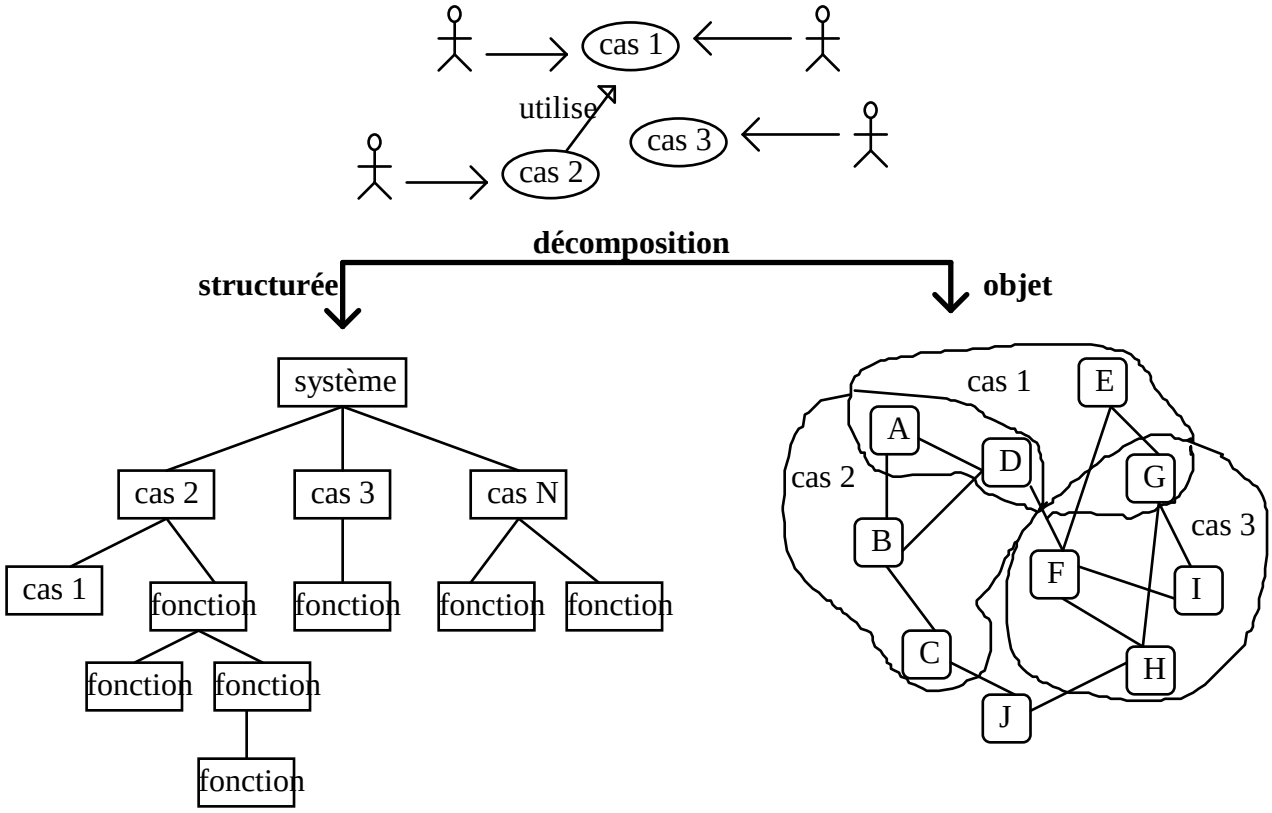
Passage des cas d'utilisation aux objets

Les cas d'utilisation sont une vue des besoins des utilisateurs. Pour passer à l'objet on associe une collaboration à chaque cas.

La collaboration décrit les objets du domaine, les connexions entre objets et les messages échangés par les objets.



Attention à ne pas décomposer les cas par structures et par fonctions, mais bien par objets et collaboration des objets.



PAQUETAGES

Paquetage

Utilité

On peut avoir à manipuler beaucoup de classes, d'interface, etc. pour résoudre un problème. Dans ce cas la visualisation simple du problème est pratiquement impossible.

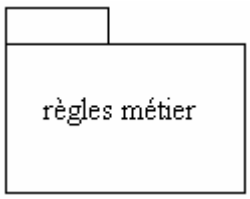
Quand on a un grand nombre de classes et de liens entre elles la visualisation s'apparente plutôt à un plat de spaghetti. C'est ce qui arrive fréquemment quand on fait du reverse ingénierie (reconstruction de diagrammes UML à partir d'un code existant).

Pour éviter ce foisonnement il existe les paquetages. Ceux-ci permettent de voir les grands points principaux d'un problème.

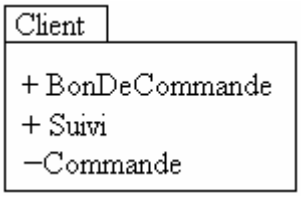
- Le paquetage contient l'ensemble d'éléments qui ont tendance à évoluer conjointement.
- Il permet d'encapsuler des éléments de modélisation.
- Un problème est composé de paquetages. Ceux-ci permettent de décrire à grands traits l'ensemble du problème.
- Les paquetages sont faiblement couplés entre eux. C.a.d. que leurs frontières sont claires et définies.
- Ils ne sont ni trop grands ni trop petits.

Représentation

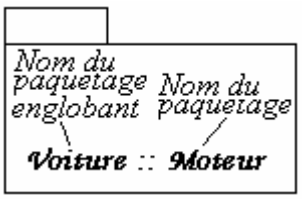
Un paquetage est représenté comme un dossier et à un nom unique.



simple



Paquetage développé



Paquetage étendu

Les éléments contenus dans un paquetage peuvent être : Des classes, des interfaces, des cas d'utilisation et même d'autres paquetages. L'élément est soit publique (+) soit privé (-).

L'imbrication de paquetages implique la décomposition hiérarchique du problème.

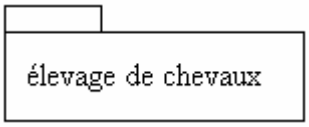
Nota : veillez à ne pas trop imbriquer les paquetages car cela devient vite illisible. 2 ou 3 emboîtements semble le maximum.

Importation et exportation

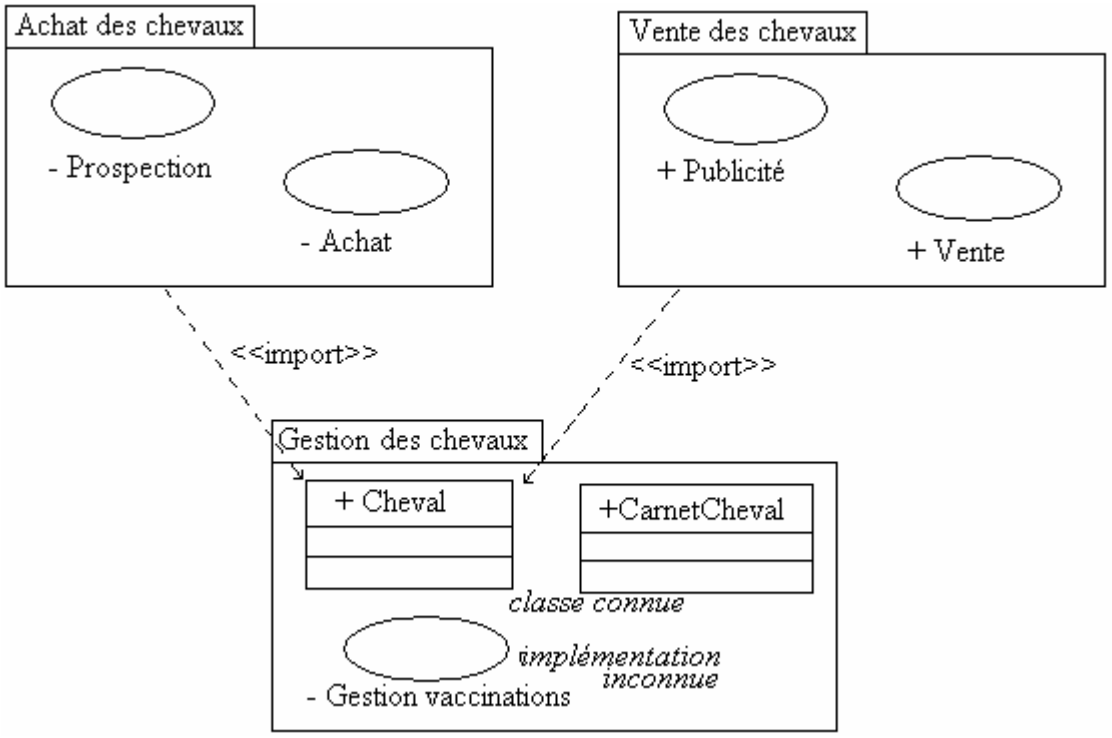
- Les parties publiques d'un paquetage sont appelées *exportations*.
- Si 1 élément est visible dans un paquetage il l'est aussi dans tous les paquetages emboîtés dans le 1er.

1 élément de la modélisation ne peut être présent que dans 1 seul paquetage. Si d'autres veulent l'utiliser il faut avoir un lien d'importation.

Ex. Le paquetage



quand on l'ouvre contient 3 paquetages.



Pour que "Achat de chevaux" et "vente de chevaux" voient la classe cheval il faut qu'ils importe "gestion de chevaux".

Modèle objet

Modèle objet

Objets et classes

- Les classes
  - Cela décrit des objets ayant des propriétés similaires, (*attributs)*, et un comportement commun, (*méthodes)*.
- Les objets
  - Les objets ont un sens précis dans le contexte étudié.
  - Chaque objet a une identité.
- Les diagrammes d'objets
  - Les diagrammes d'objets permettent par une notation graphique, de modéliser les objets, les classes et les relations qu'ils ou elles entretiennent.
  - 2 types possibles de diagramme :
    - De classes qui décrit les classes d'objets et leur relation.
      Il sert à montrer la modélisation d'un système.
    - D'instances qui décrit les relations d'un ensemble d'objets.
      Il sert à montrer des exemples.
- A un diagramme de classes correspondent une infinité de diagrammes d'instances. La distinction est artificielle et l'on mélange souvent les 2.

- Les attributs
  - Un attribut est une donnée détenue dans une classe.
- Les méthodes
  - Une méthode est une fonction qui s'applique aux attributs de la classe.

Liens et associations

- Un ***lien*** est une connexion entre des ***instances***.

  Dupond *travaille* chez Durant.

  Dupond = instance de personne.

  Durant = instance d'entreprise.

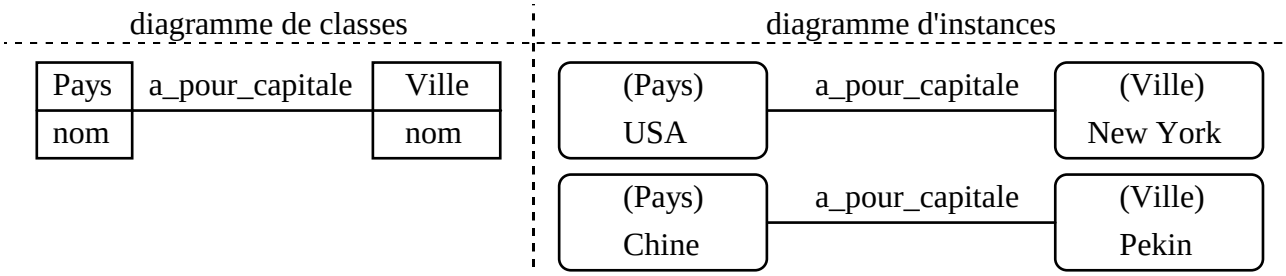
  – Un lien est une instance d'association.

- Une ***association*** est une connexion entre des ***classes***.

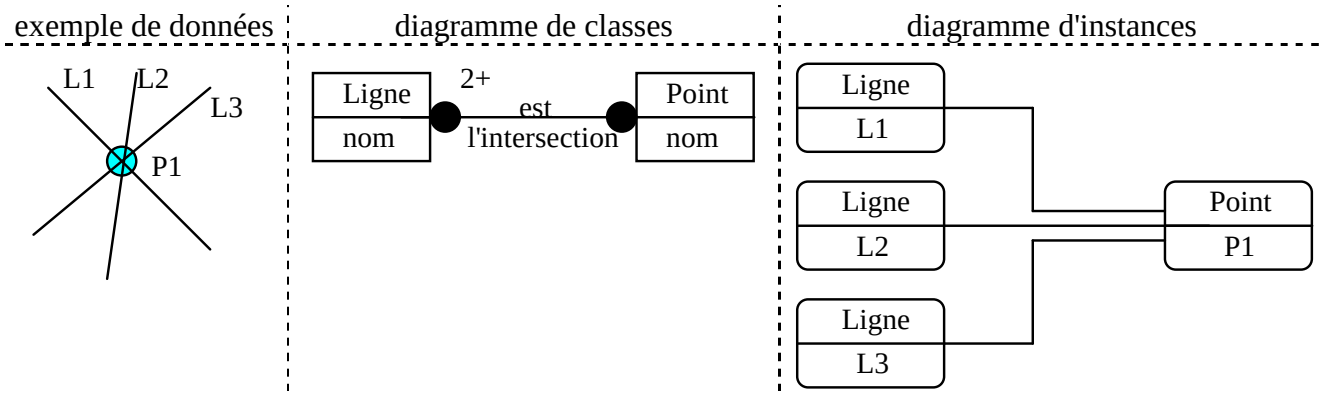
  Une personne travaille dans une entreprise

  – Bien que les associations soient bidirectionnelles, elles ne sont pas obligatoirement implémentées dans les deux directions.

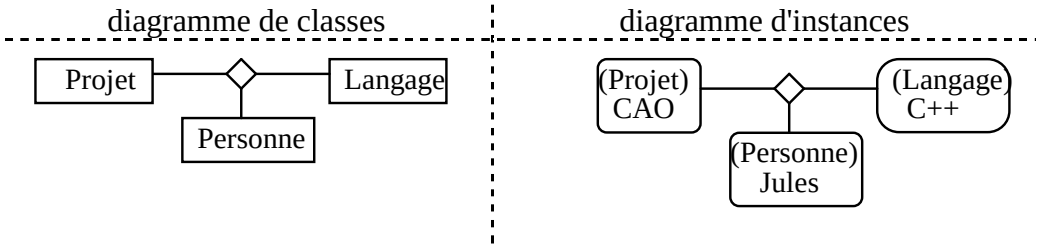
  – Une association peut très simplement être implémentée par un pointeur.



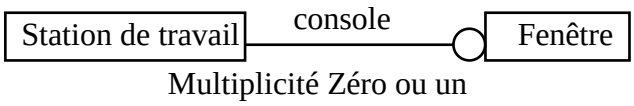
  – Dans un diagramme les associations peuvent avoir un symbole de multiplicité.



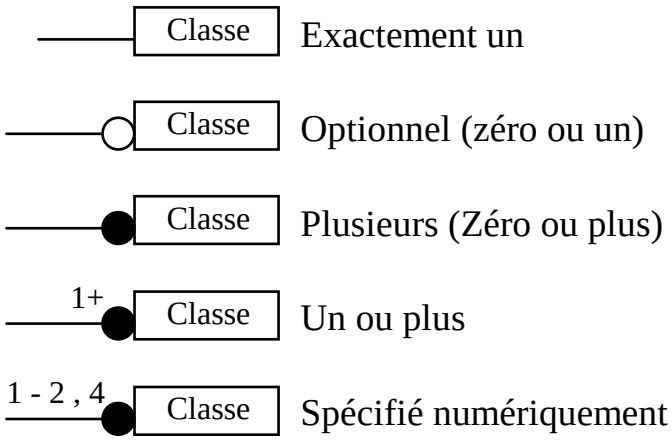
  – Une association peut être ternaire.



– La multiciplité précise combien d'instances existeront

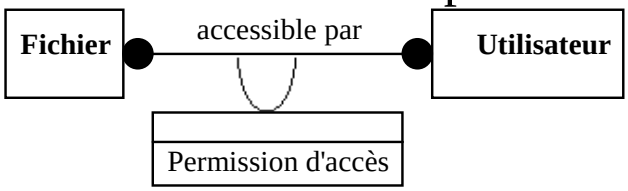


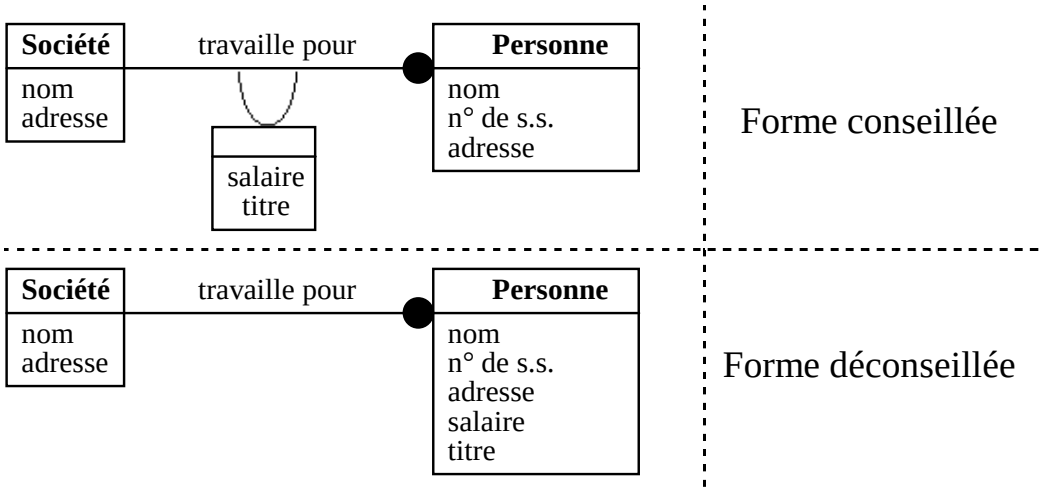
Multiplicité Zéro ou un



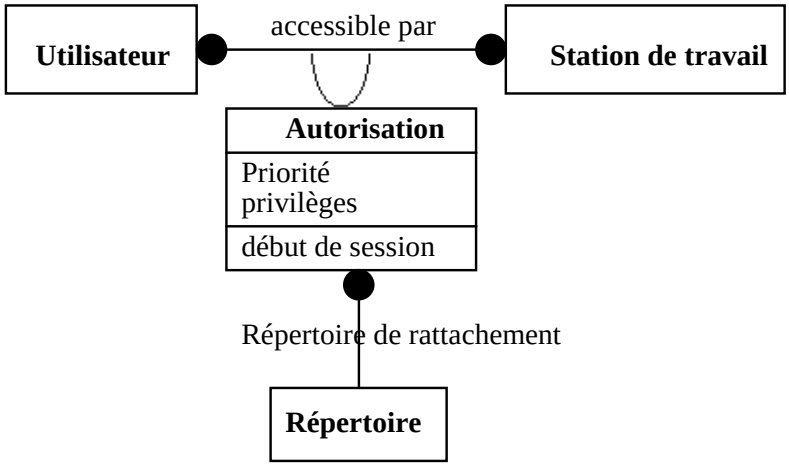
Concept évolués

- Les attributs de lien
  – Un attribut de lien est une valeur pour chaque association.





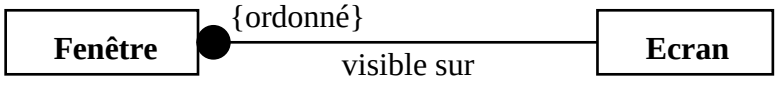
- Modéliser une association en classe.
  - Chaque lien devient une instance de classe.



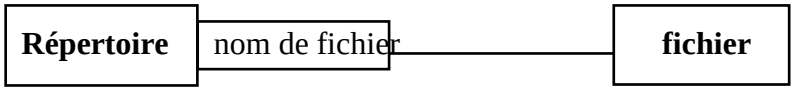
  - Il est utile de modéliser une association comme une classe quand les liens peuvent participer à des associations avec d'autres objets , ou quand les liens peuvent être sujets à des opérations.

- Nom de rôle
  - Il identifie de façon unique une extrémité de l'association.

- L'ordre
  - Les objets "plusieurs" du côté de l'association peuvent être ordonnés. C'est une contrainte que l'on impose en plus.



- La qualification
  - La qualification remplace le symbole de multiplicité.



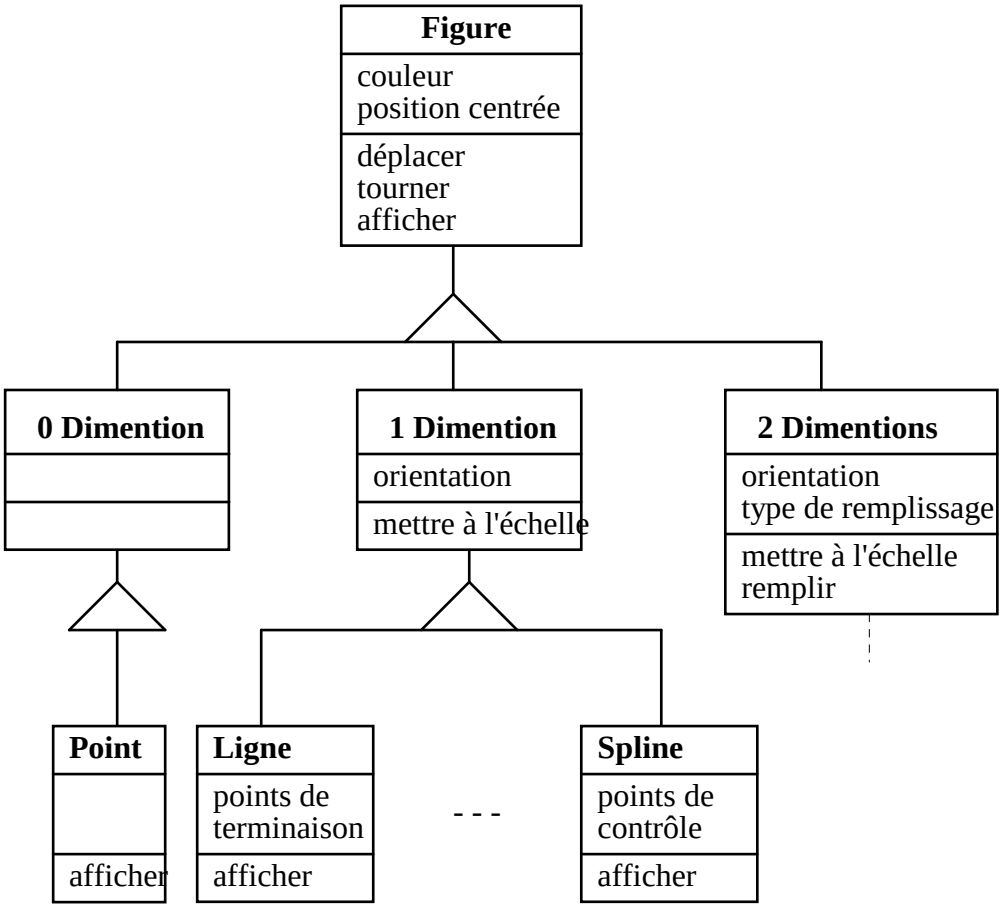
Agrégation

- L'agrégation
  - L'agrégation est une relation de type "***partie de***".

  - C'est une forme particulière d'association.
  - Décider d'utiliser l'agrégation plutôt que l'association est souvent arbitraire.
  - L'agrégation est transitive :
    A est une partie de B et B est une partie de C => A est une partie de C
  - L'agrégation n'est pas symétrique :
    A est une partie de B => B n'est pas une partie de A

Héritage

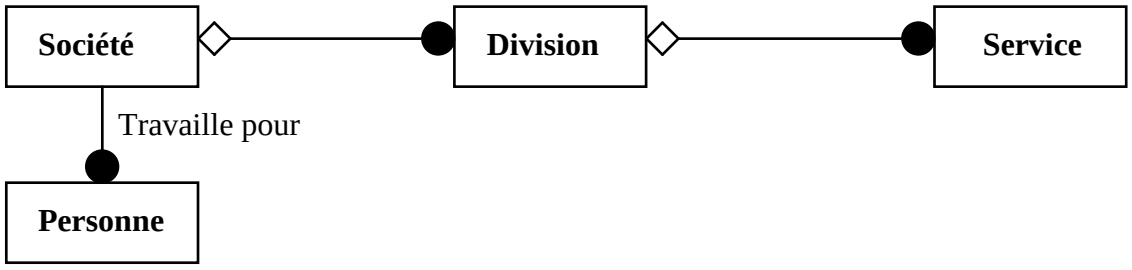
- L'héritage, ou *généralisation*, est une relation entre une classe et des versions plus affinées de la classe.
  - La classe dont on tire des précisions est appelée : *super-classe*
  - chaque version affinée est appelée : *Sous-classe* ou *classe dérivée*



Evolution du modèle objet

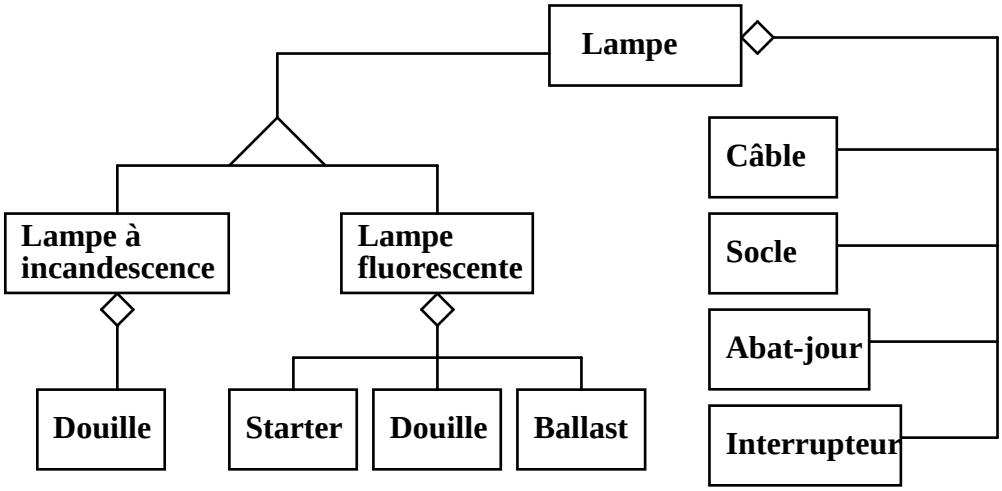
L'agrégation / l'association

- Si 2 objets sont étroitement liés par une relation composée composante, c'est une agrégation.
- Si 2 objets sont considérés comme indépendants, c'est une association.
- Pour différencier il faut se poser les questions :
  - Peut-on utiliser l'expression *partie de* ?
  - Les opérations appliquées sur le composé sont-elles appliquées automatiquement aux composantes ?
  - Les valeurs des attributs sont-elles propagées du composé vers les composantes ?
  - Y a-t-il une asymétrie intrinsèque dans l'association , dans laquelle une classe d'objets est subordonnée à une autre ?



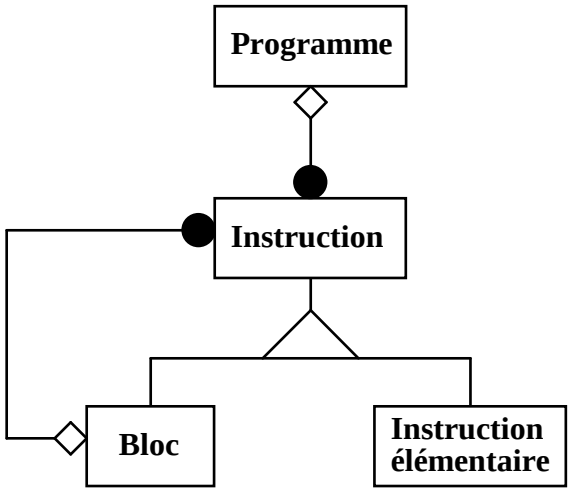
L'agrégation / l'héritage

- L'agrégation est une relation appelée *partie de*.
- L'héritage est une relation appelée *sorte de*.



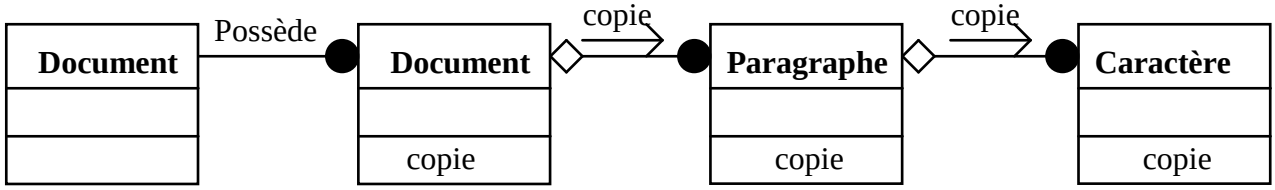
L'agrégation récursive

- L'agrégation contient une instance de même sorte d'agrégat.



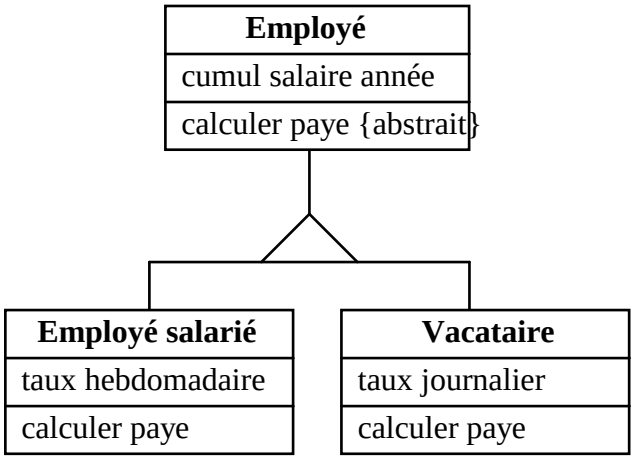
Propagation des opérations

- C'est l'application d'une opération à un réseau d'objets à partir d'un objet de départ.



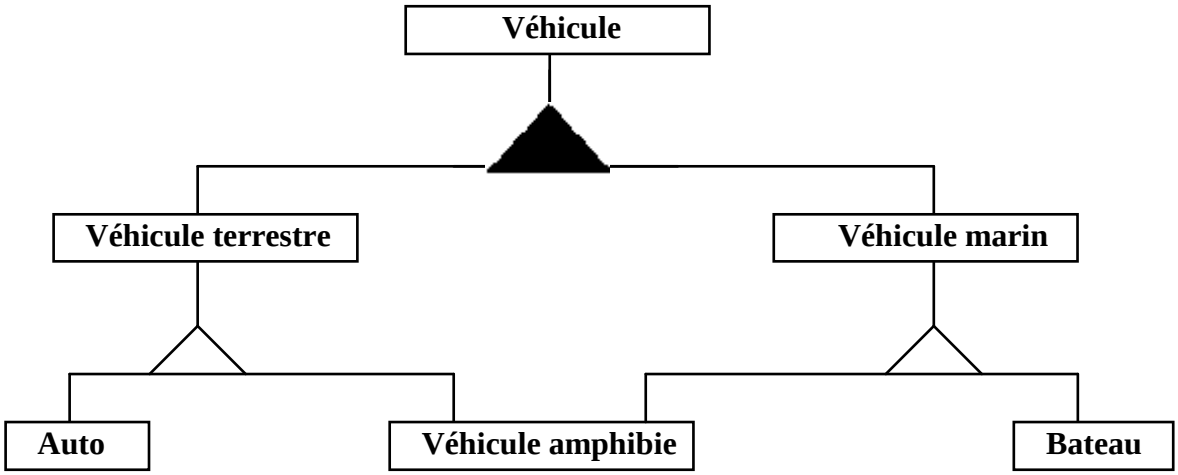
Les classes abstraites

- Une classe abstraite ne s'instancie pas. Seuls ses dérivées , si elles sont concrètes , le peuvent.

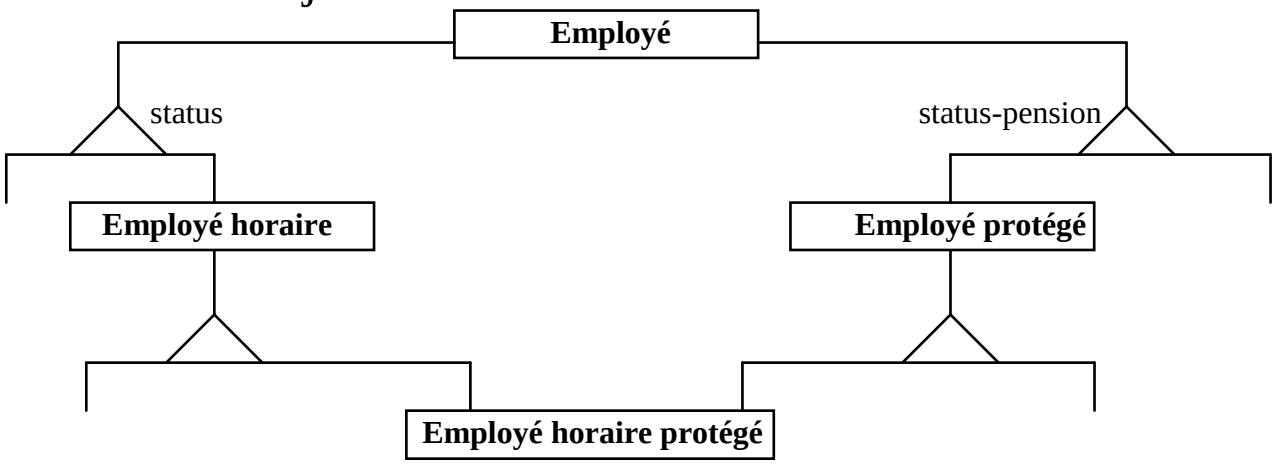


L'héritage multiple

- Avec recouvrement.



- de classes disjointes.



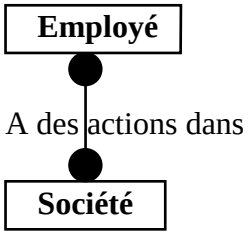
Les méta-données

- Une méta-donnée est une donnée qui en décrit une autre.
  Ex. : Un plan décrit une maison
- Une classe est une méta-donnée , en fait c'est un méta-objet.

Les clés candidates

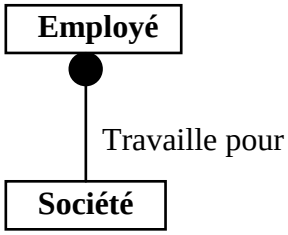
- fréquemment utilisées dans les bases de données.

Association plusieurs-à-plusieurs



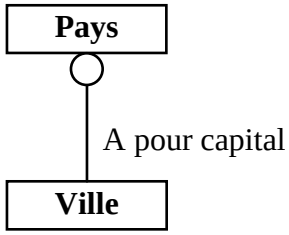
{clé candidate :
(personne , société) }

Association un-à-plusieurs



{clé candidate :
(personne) }

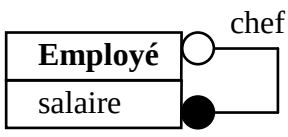
Association optionnelle -à-un



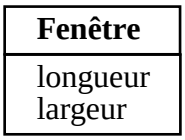
{clé candidate :
(pays)
(ville) }

Les contraintes

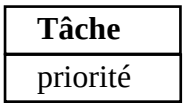
- Ce sont des relations fonctionnelles. Elles sont là pour favoriser des techniques de preuves de programme.



{ salaire ≤ salaire_du_chef }



{ 0,8 ≤ longueur/largeur ≤ 1,5 }



{ priorité ne doit pas augmenter }

Conseils pratiques

* Ne commencez pas à construire un modèle objet en *jetant pêle-mêle* classes, association et héritage.
* Efforcez-vous de garder votre modèle ***simple*** .
* Choisissez les *noms* avec soin.
* *N'enfouissez* pas des pointeurs ou des références d'objets dans des objets en tant qu'attributs. Modélisez-les plutôt comme des *associations*.
* Evitez les *associations ternaires* ou n-aires.
* N'essayez pas d'exprimer les multiplicités *trop tôt*.
* Ne *dissolvez* pas les attributs de liens dans une classe.
* Utilisez des *associations qualifiées* quand c'est possible.
* Evitez les généralisations *profondément* imbriquées.
* Reconsidérez les *associations un-à-un*. Souvent l'objet à chaque extrémité est optionnel et une multiplicité zéro-ou-un peut être plus appropriée.
* Un modèle objet a toujours besoin d'une *révision*. Il faut de multiples ***itérations*** pour tout clarifier.
* Soumettez votre modèle à des ***avis extérieurs***.
* ***Documentez*** systématiquement vos modèles objets. Une explication guide le lecteur à travers le modèle.
* Ne vous sentez pas obligés *d'éprouver toutes* les constructions de modélisation. Utilisez *seulement* ce que vous *sentez nécessaire* pour le problème que vous traitez.

Modèle dynamique

Modèle dynamique

Evénements/états

Les événements

- Un événement est quelque chose qui se produit à un moment donné dans le temps.
- Deux événements peuvent être :
  - Causals si l'un est lié à l'autre (il est la cause de l'autre).
  - Concurrents s'ils n'ont pas d'effet l'un sur l'autre.
- Un événement transporte de l'information d'un objet vers un autre.
- On regroupe les événements dans des classes d'événements, celles-ci possédant des attributs indiquant l'information dont ils sont porteurs.

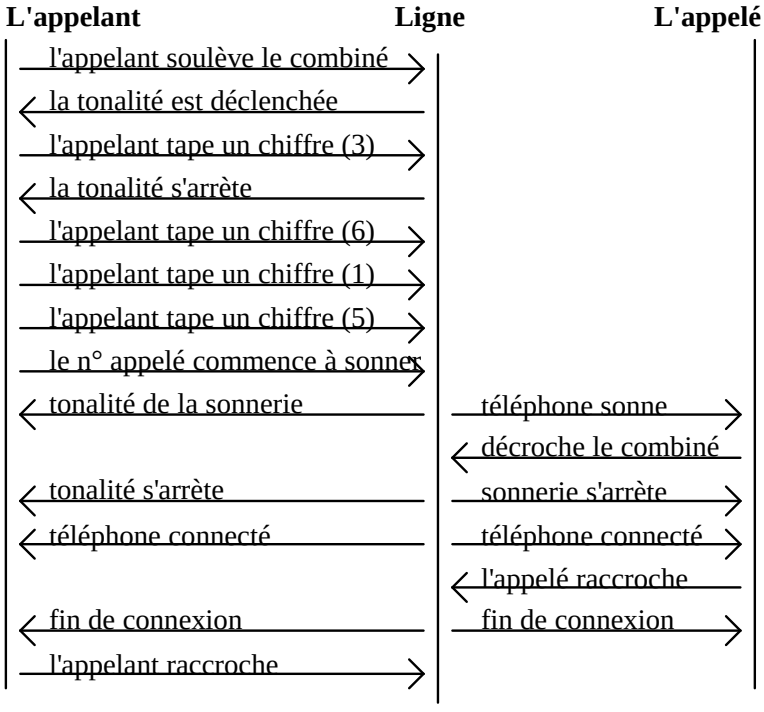
Les scénarios

- Les scénarios sont une suite d'événements.

Scénario pour un appel téléphonique

- l'appelant soulève le combiné
- la tonalité est déclenchée
- l'appelant tape un chiffre (3)
- la tonalité s'arrête
- l'appelant tape un chiffre (6)
- l'appelant tape un chiffre (1)
- l'appelant tape un chiffre (5)
- le n° appelé commence à sonner
- la tonalité de la sonnerie s'entend dans le téléphone appelant
- l'appelé répond
- le téléphone de l'appelé cesse de sonner
- la tonalité de sonnerie cesse dans le téléphone appelant
- les téléphones sont connectés
- L'appelé raccroche
- Les téléphones sont déconnectés
- l'appelant raccroche

diagramme de suivi d'événements



Etat

- Un état est une abstraction des valeurs des attributs et des liens d'un objet.
- Un état est stable entre 2 événements.
- Un état a une durée. Il occupe un intervalle de temps.
- Un état est associé à la valeur d'un objet satisfaisant à une condition.

Diagrammes d'états

- Un diagramme d'états relie des événements à des états.
- Un changement d'état est provoqué par un événement, on parle de transition.
- Un diagramme d'états est un graphe orienté, dont les noeuds sont les états et les arcs des événements.

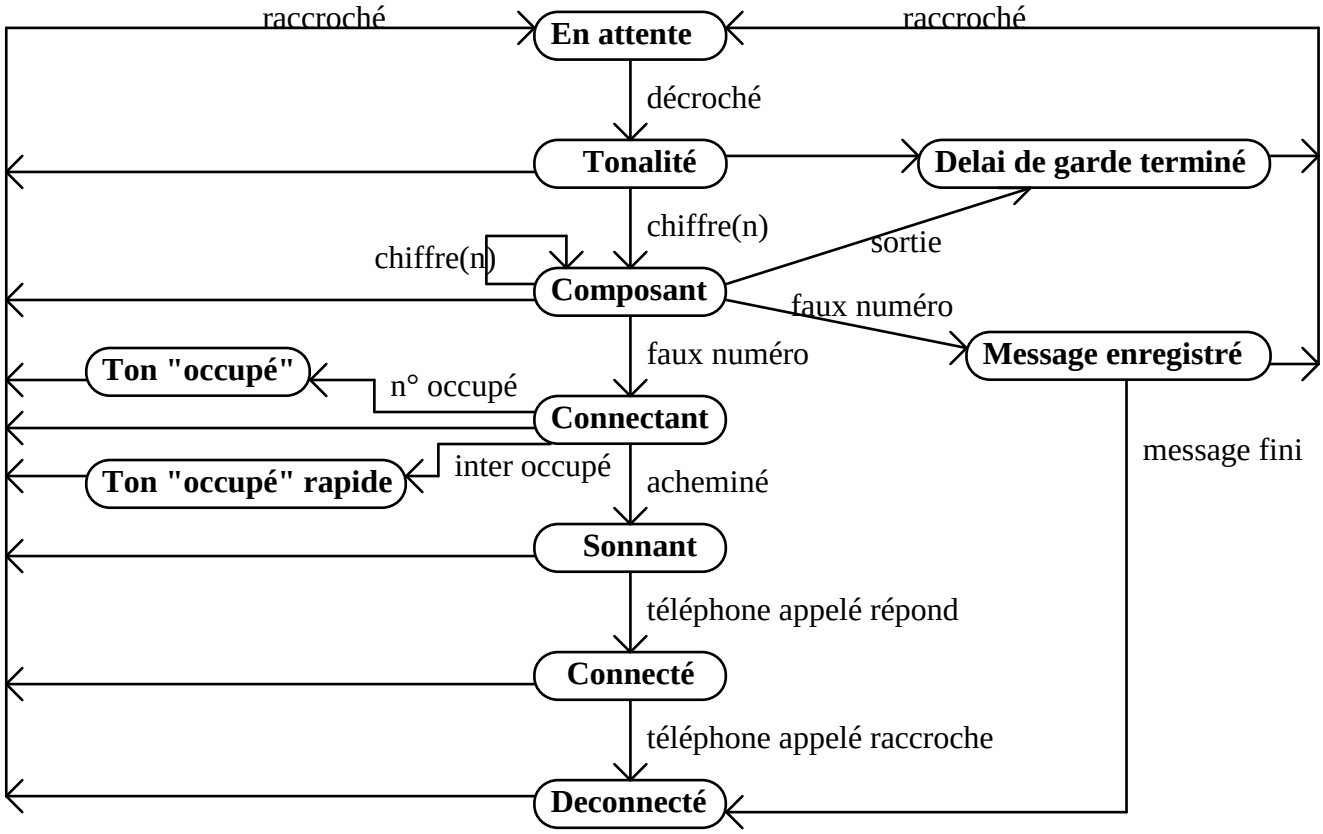
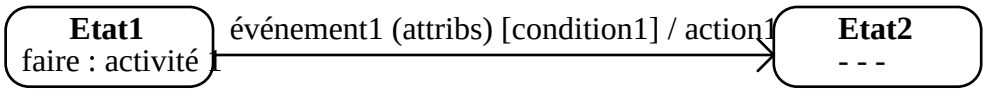


Diagramme d'états pour la ligne téléphonique

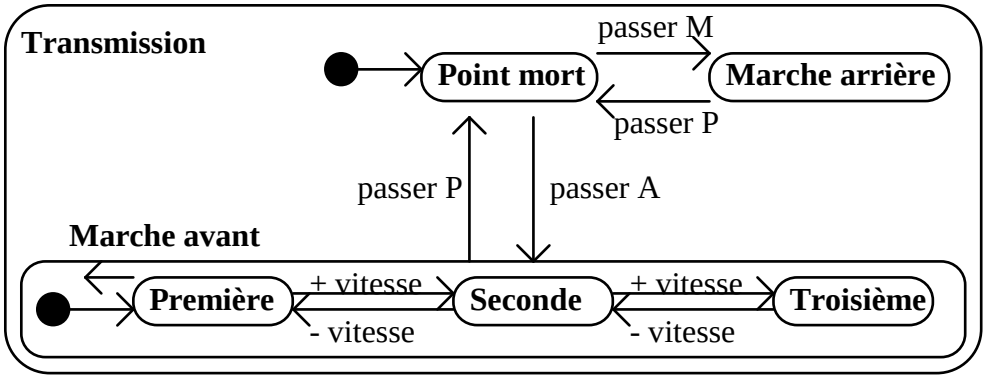
Conditions/opérations

- Une *condition* est une fonction booléenne des valeurs de l'objet.
- Une *condition* peut être valide durant un intervalle de temps.
- Une **activité** est une opération qui nécessite un *certain temps d'exécution*.
- Une **action** est une opération *instantanée*.



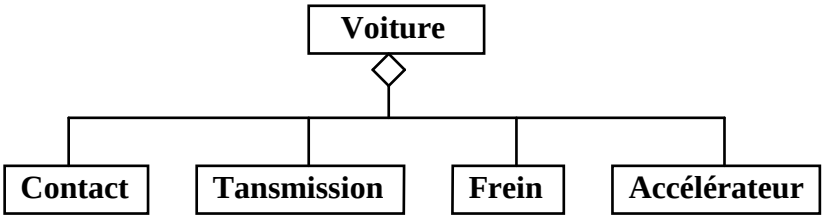
Morceaux de diagramme d'états

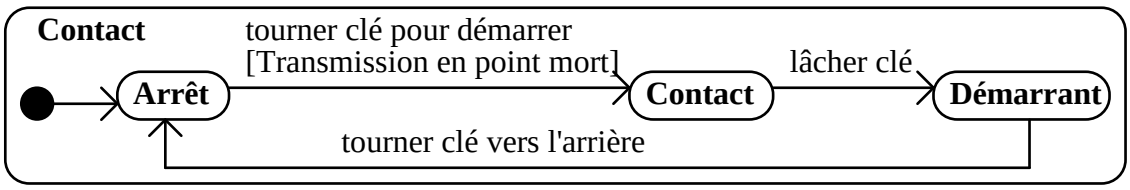
- Un diagramme d'états imbriqués est un diagramme d'états en "morceau". Chaque morceau représente une partie du diagramme total.
- Il y a une généralisation d'états possible car les états peuvent avoir des sous états qui héritent les transitions de leurs super états.

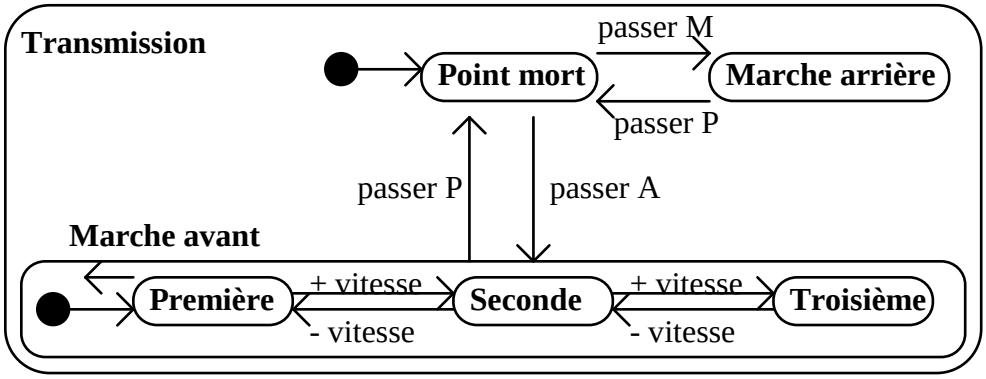


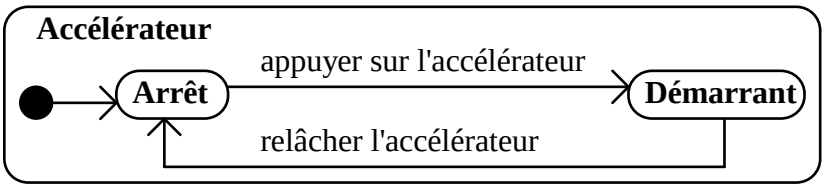
La concurrence

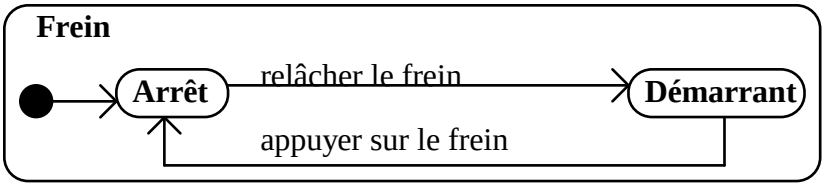
La concurrence permet de montrer dans plusieurs diagrammes tout ou partie d'un objet.





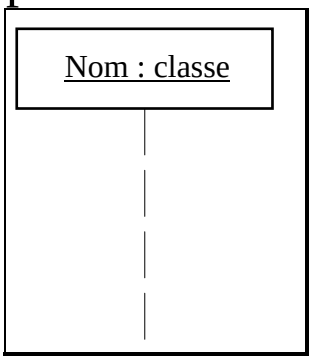






Symbolique des diagrammes de séquence

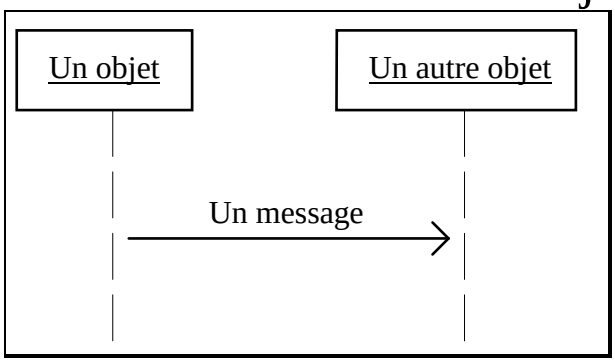
♦ représentation d'un objet



→ L'instance ou la classe est représentée par un rectangle comportant un nom

→ La barre verticale est la ligne de vie de l'objet

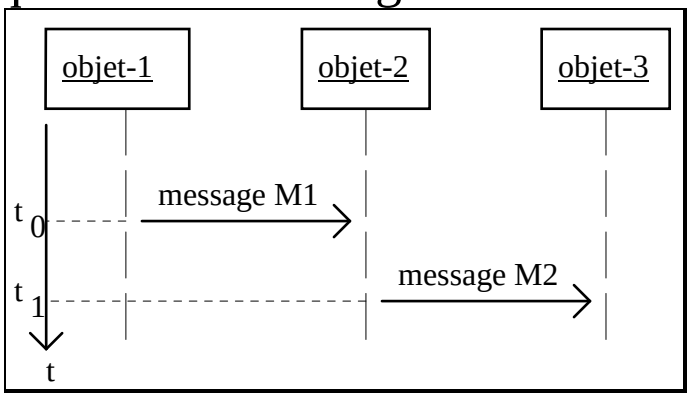
♦ communication entre les objets



→ Les instances communiquent par l'envoie de messages

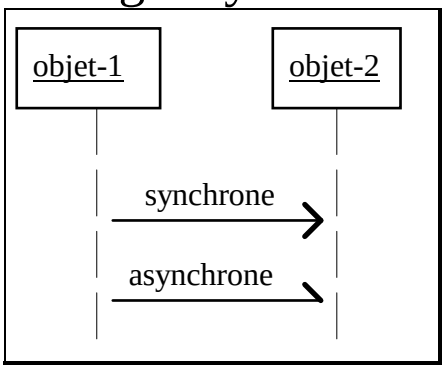
◊ Séquence de messages

♦ séquences de messages entre les objets



→ Les messages sont positionnés sur les lignes de vie, par ordre de séquence dans le temps

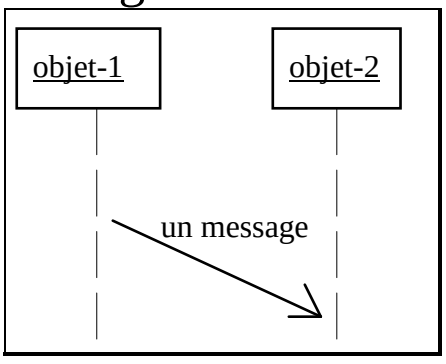
♦ messages synchrones / asynchrones



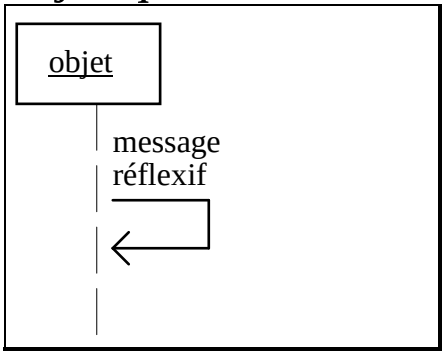
→ Lors de l'envoi synchrone l'émetteur attend la réponse

→ Lors de l'envoi asynchrone l'émetteur n'attend pas la réponse et continu son travail

♦ messages avec un délai de propagation



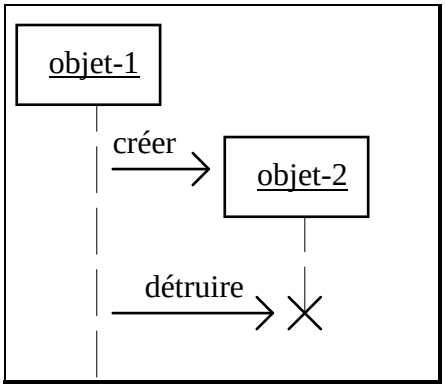
♦ objet qui s'envoie à lui-même un message



→ un message réflexif est souvent un message à un sous objet interne contenu dans l'objet

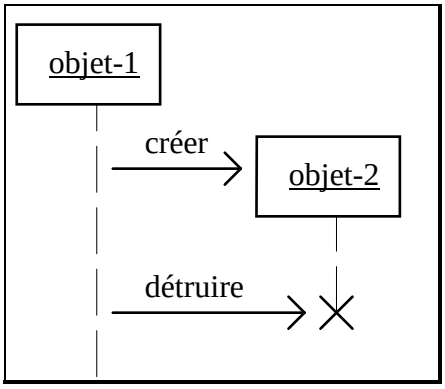
◊ Création et durée de vie

♦ création et destruction d'un objet par un message



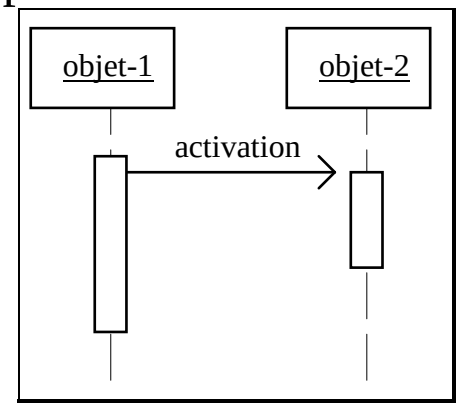
→ un message d'un objet peut créer un autre objet. Un autre message du même objet le détruira ensuite

♦ création et destruction d'un objet par un message



→ un message d'un objet peut créer un autre objet. Un autre message du même objet le détruira ensuite

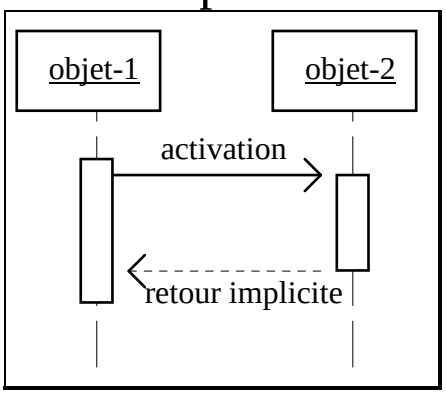
♦ période d'activité d'un objet



→ un message d'un objet 1 active l'objet 2 pendant un certain temps. L'activité sur l'objet 1 continue après la fin d'activité sur l'objet 2.

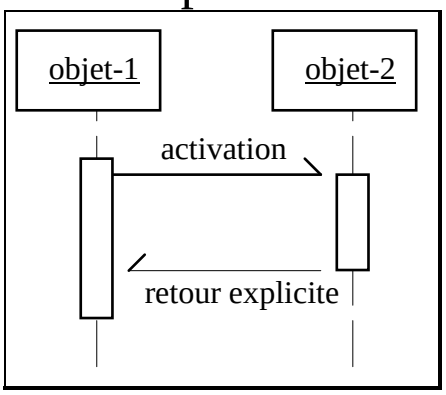
◊ Retour de message

♦ retour implicite



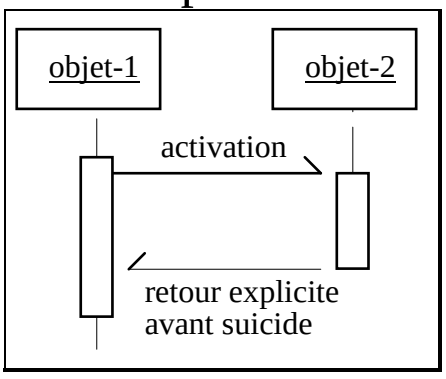
→ Le retour du message est implicite dans le cas d'un message synchrone

♦retour explicite



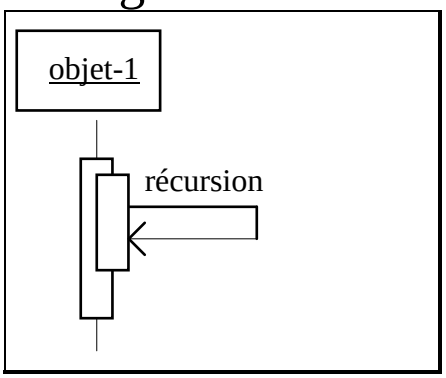
→ Le retour du message est explicite quand il s'agit de message asynchrone

♦retour explicite avant suicide



→ Le retour du message est explicite et la ligne de vie de l'objet s'arrête

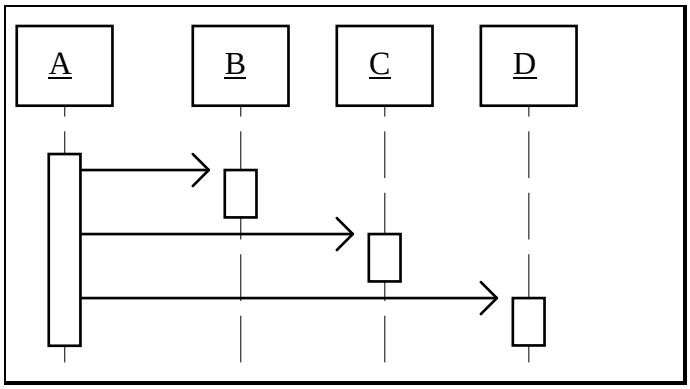
♦message récursif



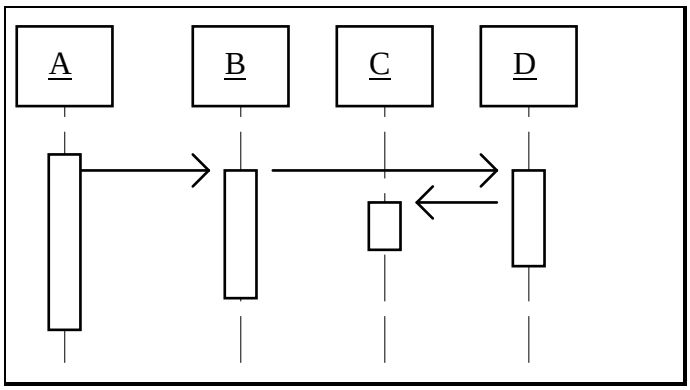
→ Le message est récursif

◊ Structures de contrôle

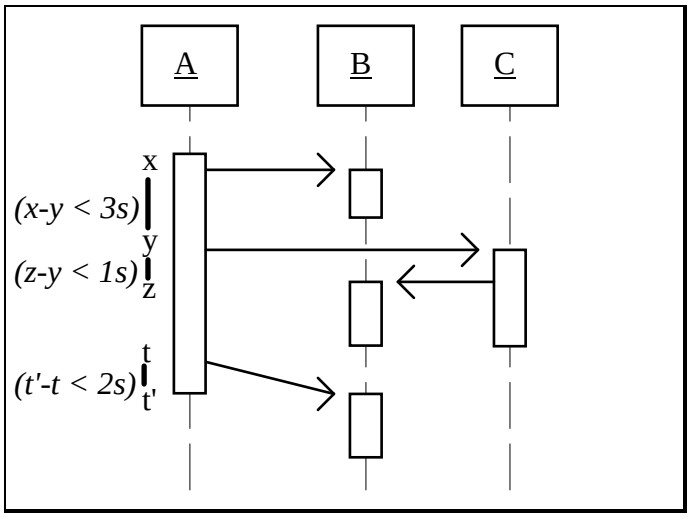
♦ mode de contrôle



→ Le contrôle est centralisé par A

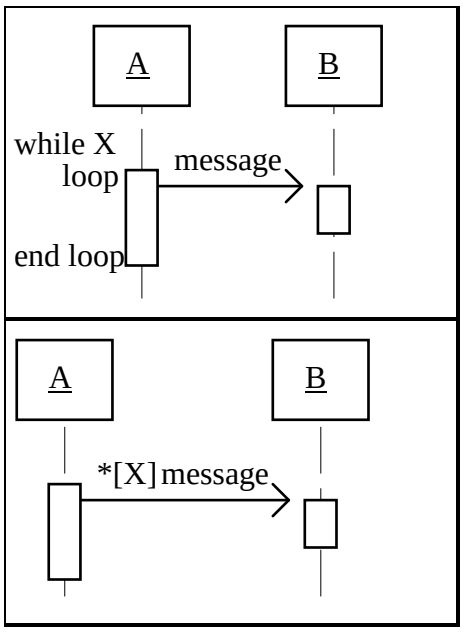


→ Le contrôle est décentralisé :
A envoie à B qui envoie à D qui envoie à C



→ On rajoute des contraintes de temps dans les séquences
(sys. temps réel)

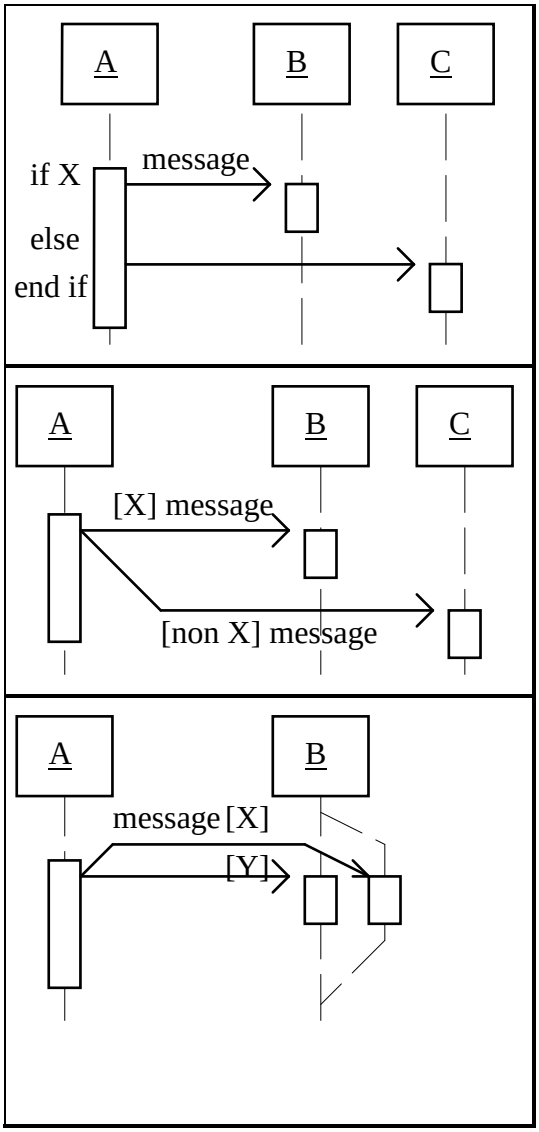
♦ boucle



→ Représentation d'une boucle while

→ La même boucle while , représentée de manière plus symbolique

♦ Test



→ Représentation d'un conditionnel if

→ Le même test if , représentée de manière plus symbolique

→ Le même test if , mais c'est le destinataire qui fait l'alternative. La ligne de vie de B est dédoublée. (La condition est placée après le message.)

Modèle objet et modèle dynamique

- Le modèle dynamique spécifie les séquences de modification des objets du modèle objet.
- Les états sont les classes d'équivalences des valeurs de liens et d'attributs de l'objet.
- Les événements sont les opérations sur le modèle objet.
- La structure du modèle dynamique est contrainte par la structure du modèle objet.
- Une hiérarchie des états d'un objet correspond à une hiérarchie de restriction de la classe de l'objet.
- La concurrence des états vient de :
  - L'agrégation d'objet ; chaque classe agrégée fournit un diagramme concurrent.
  - L'agrégation à l'intérieur d'un objet ; chaque attribut et lien définit des sous états concurrents.
  - Un objet est parfois obligé d'exécuter des actions concurrentes.
- Une hiérarchie d'événements est indépendante de la hiérarchie de classes.

Conseils pratiques

* Ne construisez un diagramme d'états que pour les classes ayant un comportement dynamique *significatif*.
* Sur les divers diagrammes, vérifiez l'uniformité des événements partagés.
* Utilisez des *scénarios* pour vous aider à *démarrer*.
* Ne considérez que les attributs pertinents pour définir un état.
* Considérez les besoins de l'application quand vous prenez des décisions quant à la granularité des événements et des états.
* L'application vous guide dans le choix entre actions et activités (instantanéité ou durée).
* Quand un état a plusieurs arcs d'arrivée et que tous ces arcs provoquent la même action, placez les actions à l'intérieur des états précédés par un événement *entré* au lieu de les énumérer sur les arcs. Idem pour les événements *sortie*.

.....

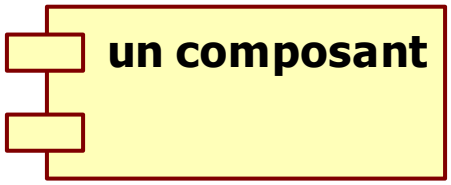
Composants

Les composants

Les briques de bases des applications informatiques sont les librairies et autres dll. Ce sont les composants.

- Ils ont un nom
- Ce sont des interfaces
- Ils sont réutilisables
- C'est une partie physique remplaçable d'une application

Ex.

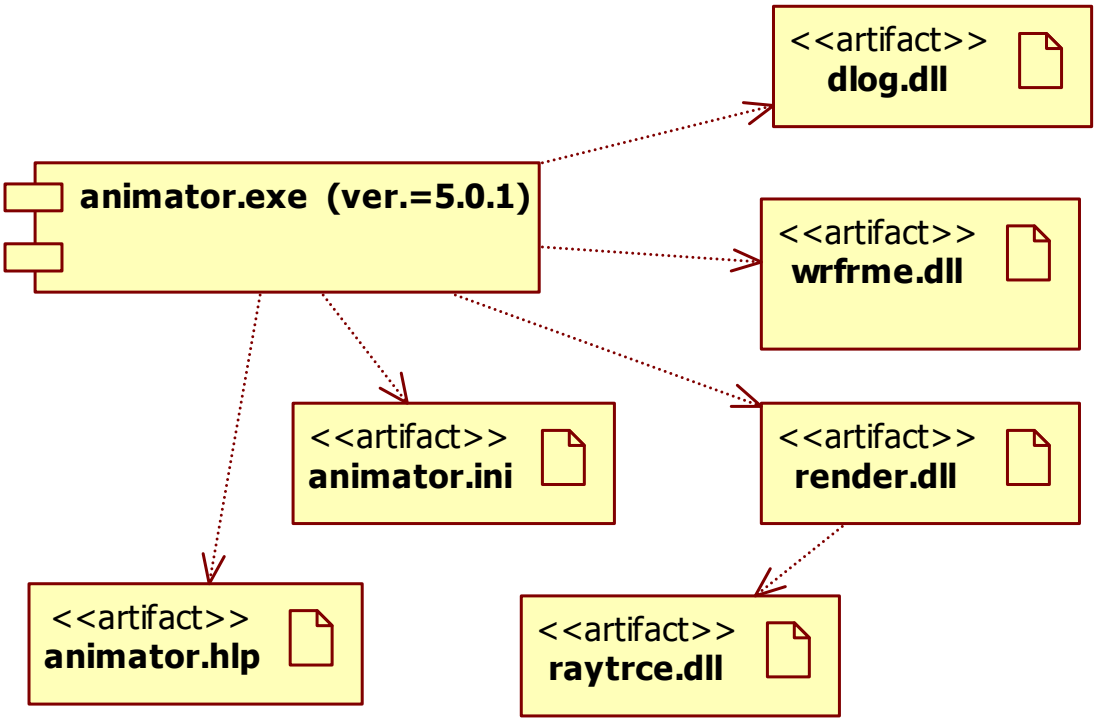


Un composant est constitué de : classes, composants, et autres fonctions.

La différence entre classe et composant c'est qu'une classe contient des attributs et des méthodes. Un composant n'a que des opérations que l'on peut atteindre par son interface.

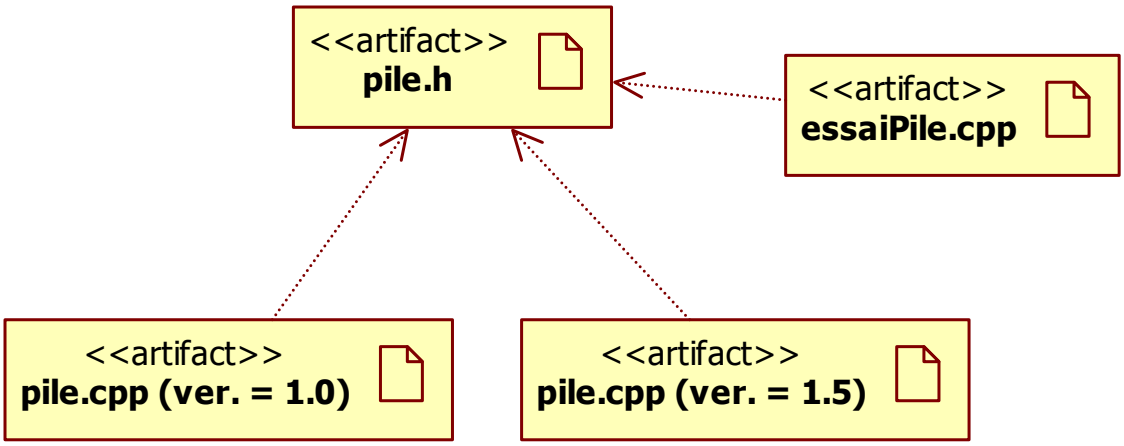
Les composants sont construits avec des librairies des fichiers d'exécutables et des fichiers d'aides.

Ex. un programme qui est fait à partir de librairies et de fichiers d'init et d'aide.



On peut aussi modéliser le code source d'une application.

Ex. de programme d'essai d'une pile.



C'est surtout ce modèle là qui est important.

Lorsqu'on a une application complexe on est très heureux de savoir de façon certaine et simple de quoi elle est composée. Le modèle de composant est une manière simple et efficace de montrer comment et avec quoi le programme est construit.

C'est ce qui devrait toujours exister dans la documentation de conception des programmes et qui est malheureusement encore trop rarement fait.

C'est la marque d'un bon developpeur que de rendre un executable qui marche et avec sa documentation de réalisation.

Exemple

Exemple simple

Nous allons étudier un système de gestion automatique d'aquarium.

Un aquarium est un petit univers aquatique clos qui doit permettre de maintenir artificiellement, et de manière la plus fidèle possible, la vie de la flore et de la faune.

Pour cela il faut assurer pour les poissons et les plantes :

-un chauffage constant à une valeur de consigne fixée.

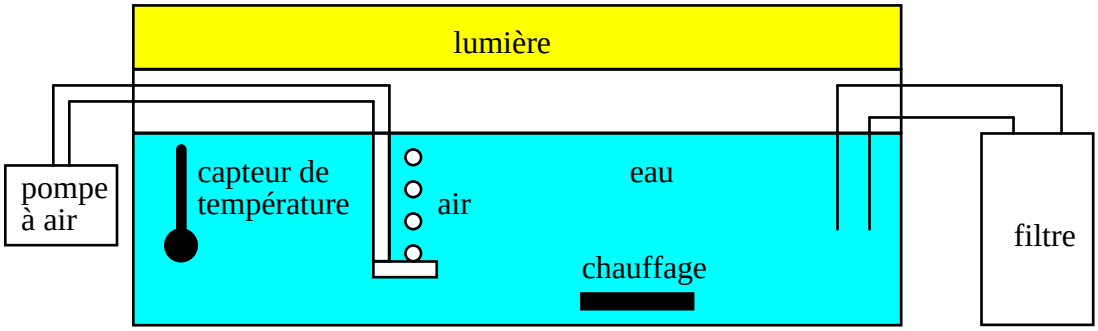
-un éclairage durant un certain nombre d'heures.

-un mécanisme de filtrage qui évacue les impuretés de l'eau.

-une aération de l'eau pour la respiration de l'oxygène des poissons.

Il faudra que notre système informatique soit assez souple afin de pouvoir rajouter par la suite d'autres fonctionnalités, tel que la distribution automatique de nourriture, etc.

Un aquarium du commerce est fait, à peu près, avec les éléments suivant :



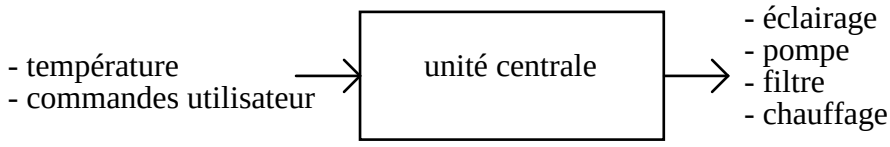
Nous voulons la même chose mais contrôler les éléments par ordinateur.

On doit donc gérer :

- Un dispositif de chauffage.

- Un dispositif d'éclairage.
- Un dispositif de pompe/filtration.
- Une interface de dialogue avec l'utilisateur.

Les données sont :



Analyse

On peut commencer à énumérer les fonctions qui vont nous être nécessaires.

- Centrale de gestion.
- Dispositif de chauffage.
- Dispositif d'éclairage.
- Dispositif de filtrage.
- Saisie des commandes

Décomposition des fonctions.

- Dispositif de chauffage.

- Il est constitué : d'une résistance chauffante qui est commandée par la centrale de gestion, d'un capteur de température qui envoie ses mesure à la centrale et d'une consigne fixée dans la centrale.

- Dispositif d'éclairage.
  - Il est constitué : d'une lampe qui est commandée par la centrale de gestion et d'une consigne de mise en marche et d'arrêt fixée dans la centrale de gestion.
- Dispositif de pompe/filtrage.
  - Il est constitué : d'une pompe qui fonctionne sans arrêt, sauf quand on veut la nettoyer et d'une pompe à air qui fonctionne tout le temps.
- Saisie des commandes.
  - On suppose pour faciliter l'analyse que c'est fait au moyen d'un clavier classique avec un dispositif d'affichage tout aussi classique.
- Centrale de gestion.
  - On va supposer pour l'instant que c'est un PC. Il faudra certainement simplifier et considérer que c'est un système particulier fait avec une carte électronique spécialisée.
  - Elle est constitué de :
    * Une gestion de filtre. Il faut pouvoir l'arrêter et le remettre en marche.
    * Une gestion de température. Elle contient une consigne de température qui est fixe.
    * Une gestion d'éclairage. Il faut un **temporisateur** afin d'arrêter ou de mettre en marche les lampes.
    * Une interface de dialogue avec l'utilisateur qui va être suivie d'un interprète de commandes, afin de pouvoir saisir les consignes voulues (nous n'en ferons pas l'analyse ici).

nota : Nous venons de découvrir qu'il faudra une horloge. On devra ***réifier*** le temps.

Identification des attributs

- Chaque dispositif possède une valeur de fonctionnement : est-il en marche ou arrêté ?
- Le chauffage a une valeur de consigne.
- L'éclairage a une heure de mise en marche et une heure d'arrêt.

Identification des opérations

- Les gestions de dispositif doivent avoir les opérations on/off.
- La gestion de température doit lire la température issue du capteur, et celle qui est fixée comme consigne.
- La gestion d'éclairage doit comparer son temps de marche et son temps d'arrêt avec l'horloge.

Identification des classes

Il semble naturel de considérer les éléments physiques comme des classes *a priori*. On aurait donc les classes suivantes :

- Température
- Eclairage
- Pompe

Il manque la notion de temps qu'il faut réifier.

- Temps

Cette liste est cependant pauvre et peu détaillée. Il faut beaucoup préciser.

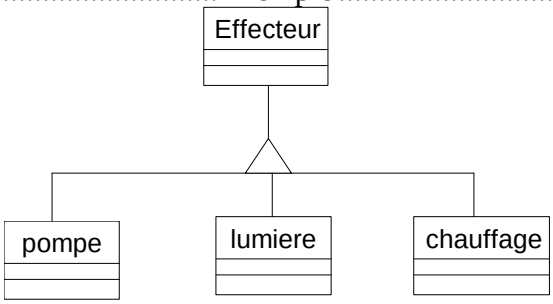
Effecteurs

Nous avons 3 types de comportement :

Les *constantes* : Les pompes marchent toujours.
Les *booléens* : La lumière s'allume et s'éteint à des moments précis.
Les *linéaires* : La température doit être suivie au plus près d'une consigne fixée.

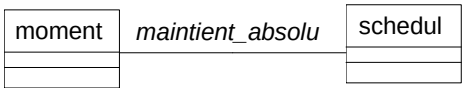
On peut regrouper les *constantes* et les *booléens* en disant que la *constante* c'est un *booléen* qui ne change pas d'état. Il ne reste que 2 types de comportement.
Si l'on considère que la température, c'est allumer ou éteindre la résistance chauffante placée dans l'aquarium, on se ramène au cas *booléen*. Il ne reste donc plus qu'un seul type de comportement. Il faut néanmoins donner une raison de changer d'état.

On a découvert une hiérarchie de classes qui n'était pas évidente.



Temps

Le temps est constitué de tops d'horloge. Ces tops vont générer une date (heure + date), qui va être l'objet avec lequel on va travailler et que nous appellerons *moment*. Il y a aussi un système de "scheduleur" qui va permettre de parcourir les différentes instances des effecteurs afin de détecter si elles doivent réagir. Ce *scheduleur* doit aussi maintenir une instance d'une date du système



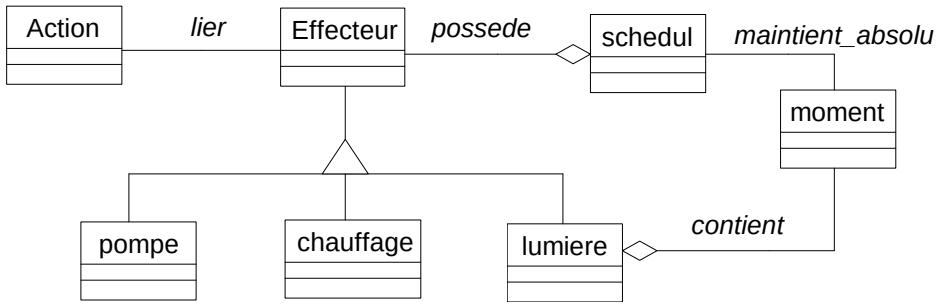
Il semble maintenant que l'on ait toutes les classes nécessaires pour gérer notre aquarium. Il manque probablement l'action que doit faire un effecteur. Allons-nous la réifier en tant que classe ou en faire une méthode des classes effecteurs, c'est secondaire pour l'instant.

Liaison

Il existe un lien entre le scheduleur et les différents effecteurs. Le scheduleur a une liste des instances d'effecteurs afin de pouvoir les interroger les uns après les autres.

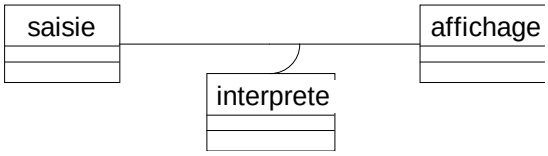
De la même manière un effecteur a une liste de moments où il doit faire une action.

Modèle objet final



On a créé une classe *Action* en supposant qu'on sera capable d'instancier autant d'actions que nécessaire. Il est possible que cette classe ne soit jamais réalisée. Si l'on suppose que les dérivées de *Effecteur* ont une action mémorisée comme une méthode interne à la classe, alors la classe *Action* n'est pas utile. Par contre si l'on suppose que l'action est reliée à un périphérique spécifique et qu'elle en fait partie, alors il est normal de lier *Action* et *Effecteur*.

Jusqu'ici nous n'avons pas représenté le modèle objet de l'interpréteur de commandes. Nous pouvons le décrire comme étant un système de saisies, d'affichage et de vérification de syntaxe.



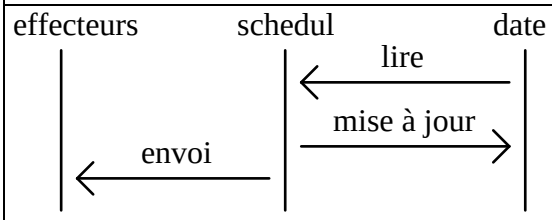
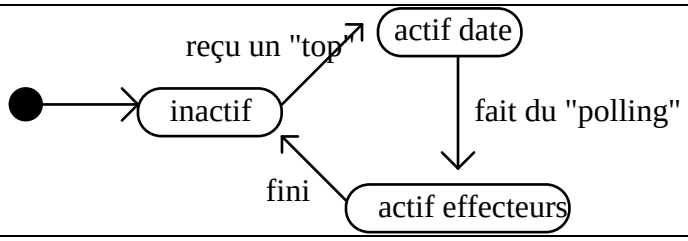
Nous ne dirons rien de plus en invitant les lecteurs à compléter cette facette de la gestion d'aquarium. Il faut en particulier bien décrire le modèle dynamique, pour voir le "status" des consignes que l'on veut transmettre.

Scheduleur.

A la réception d'un top d'horloge, il doit parcourir la liste des effecteurs et leur envoyer la date qu'il maintient à jour.

Scénario

Quand il reçoit un top il :
- met à jour la date
- parcourt la liste des effecteurs et leur envoie la date

Diagramme d'événements	Diagramme d'états
	

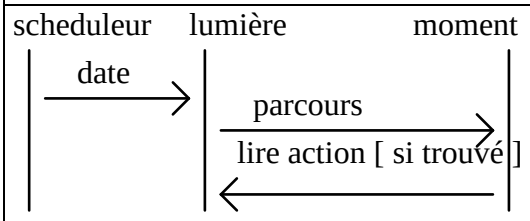
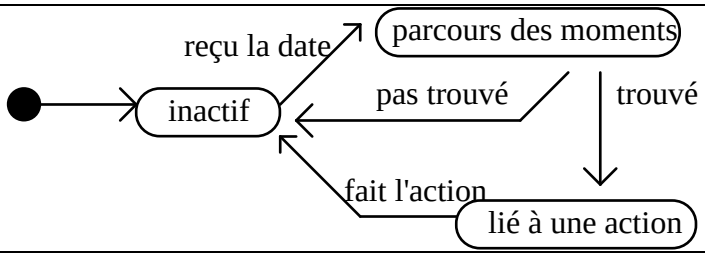
Lumières

A l'initiative du scheduleur le gestionnaire de lumière va recevoir la date du système.

Scénario

Quand il reçoit la date :
- il parcourt l'agrégation de moments.
- quand il a trouvé un moment égal à la date il effectue l'action reliée au moment.
- il "marque" le moment dans l'agrégation afin de pouvoir "partir" de celui-ci lorsqu'il recommencera.

(On voit se dessiner une structure de données de type liste circulaire pour l'agrégation de moments).

Diagramme d'événements	**Diagramme d'états**
	

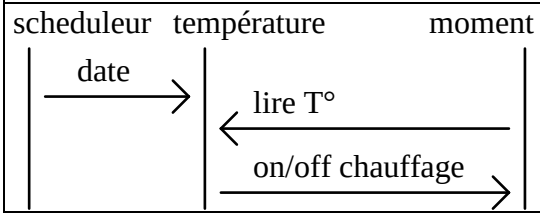
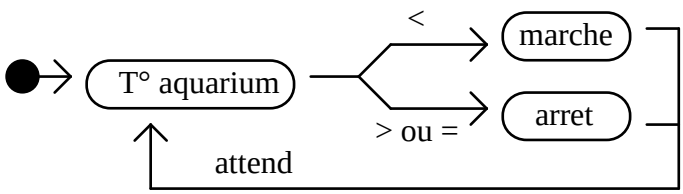
Température

Quand le scheduleur le "réveille" il teste si la température de l'aquarium est < à une consigne, si oui il met le chauffage en marche, sinon il arrête le chauffage.

Scénario

Quand il reçoit la date il :

- teste la valeur de T° de l'aquarium par rapport à sa consigne

Diagramme d'événements	Diagramme d'états
	

nota : Cette méthode de régulation par tout ou rien est très frustre. Il y a d'autres moyens plus fins de réguler une température , PID[3] , filtres numériques auto adaptatifs, ou logique floue , qui sont plus adaptés à ce type de problème. Mais cela sort du cadre d'un cours de méthodologie. Si l'on devait effectivement réaliser cette fonction, il faudrait aller consulter un spécialiste de la régulation linéaire.

[3] La régulation *Proportionnelle Intégrale Dérivée* permet de minimiser les variations qui se produisent lors d'une régulation par une méthode d'atténuation des oscillations , donc action équivalente à un filtrage sur un signal.

Conclusion

On a réussi à modéliser la gestion automatique d'un aquarium sans difficulté. De plus, le fait de le modéliser permet de bien comprendre le fonctionnement de l'automatisme de régulation. Si l'ajout d'un besoin tel que la distribution automatique de nourriture se présente, nous pourrons reprendre les différents modèles et l'intégrer sans tout "casser", ce ne sera qu'une fonctionnalité en plus , donc le fonctionnement correct sera préservé.

Bibliographie

Bibliographie

The Design and Evolution of C++ Bjarne Stroustrup
Addison-Wesley 1994

La qualité en C++ Philippe Prados Eyrolles 1996

Object-oriented modeling and Design James Rumbauch
Prentice-Hall 1991
Michael Blaha
William Premerlani
Frederick Eddy
William Lorensen

OMT James Rumbauch et al.
Prentice-Hall 1995
1. Modélisation et conception orientées objet
Masson
2. Solution des exercices
Masson 1996

Unified Method ver 0.8 Grady Booch Rational 1995
James Rumbauch

Modélisation objet avec UML Pierre-Alain Muller Eyrolles 1998

Intégrer UML dans vos projets Natalie Lopez
Eyrolles 1998
Jorge Migueis
Emmanuel Pichon