Ecole Nationale d'Ingénieurs de Brest

— Cours d'Informatique S6 —

Modèle pour l'Ingénierie des Systèmes (UML)

Cédric BUCHE

buche@enib.fr

version du 29 janvier 2014

Informatique S6-MIS

Modèle pour l'Ingénierie des Systèmes

— UML —

Cédric Buche

École Nationale d'Ingénieurs de Brest (ENIB)

28 janvier 2014

Ces notes de cours accompagnent les enseignements d'informatique du 6^{ieme} semestre (S6) de Modèle pour l'Ingénierie des Systèmes (MIS) de l'Ecole Nationale d'Ingénieurs de Brest (ENIB : `www.enib.fr`). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

Une partie de ce document est librement inspiré de l'ouvrage de Laurent Audibert. Une partie des supports ont été rédigés par Pierre Chevaillier.

Table des matières

Cours

1 Diagramme d'états-transitions

Sommaire

1.1 Introduction

1.1.1 Définitions — rôles

Les diagrammes d'états-transitions UML proposent une *vue dynamique* du comportement d'un classifier (évolutions possibles des états).

Objectifs Il s'agit de décrire le comportement interne d'un classifier :
- ▷ Ensemble des états possibles (pertinents)
- ▷ Possibilités de changements d'état sur occurrence d'événement

Origine Les diagrammes d'états-transitions UML sont fondés sur les *Statecharts* (David Harel, 1987). Plus précisément, les diagrammes d'états-transitions UML utilisent des machines à états.

Machine à états Le comportement ne dépend pas uniquement des entrées du système, mais aussi de son « histoire », son état interne

Type de modèle Les diagrammes d'états-transitions UML permettent de modéliser un système discret :
- ▷ comportement caractérisable (= observable) par une succession d'états
- ▷ avec changement d'état discontinu (= discret)

Utilisation pratique Les diagrammes d'états-transitions UML permettent classiquement de préciser le comportement interne d'une instance de classe, mais ils peuvent être utilisés aussi pour d'autres éléments comme une méthode ou un cas d'utilisation. Les diagrammes d'états-transitions est le seul diagramme UML à offrir une vision complète de l'ensemble des comportements de l'élément auquel il est attaché. En effet, un diagramme d'interaction (séquence / communication) n'offre qu'une vue partielle correspondant à un cas d'utilisation.

1.1.2 Représentation graphique

Un automate à états finis est graphiquement représenté en UML par un graphe comportant des états, matérialisés par des rectangles aux coins arrondis, et des transitions, matérialisées par

des arcs orientés reliant les états entre eux.

1.1.3 Exemple : télérupteur dans une maison

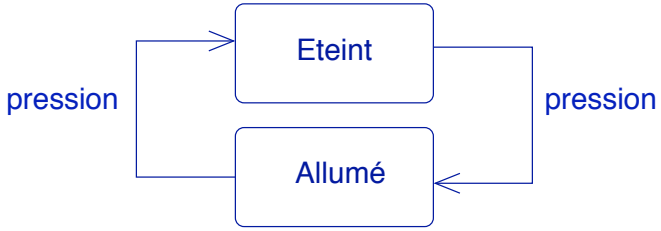
Lorsque l'on appuie sur un bouton poussoir, la réaction de l'éclairage associé dépend de son état courant (donc de son historique) :

- ▷ si la lumière est allumée, elle s'éteint,
- ▷ si elle est éteinte, elle s'allume.

Les éléments du comportement du système :

- ▷ 2 états : *Allumé* et *Éteint*
- ▷ 2 transitions : *Allumé* $\mapsto$ *Éteint* et *Éteint* $\mapsto$ *Allumé*
- ▷ 1 événement : *pression* sur un bouton poussoir

Figure 1.1. *Un automate d'états fini simple*



1.1.4 Interprétations du modèle

Deux interprétations sont possibles :

1. Machine à état comportementale.
   Il s'agit de spécifier le comportement d'un classifier, instance d'une classe, cas d'utilisation, collaboration, méthode : **ce que fait un classifier** quand on le sollicite. Graphiquement, il faut indiquer **stm** {behavior}.
2. Machine à état de protocole.
   Il s'agit de spécifier les séquencements « légaux » de sollicitations : **comment utiliser un classifier**, sans effet sur une instance du classifier. Graphiquement, il faut indiquer **stm** {protocol}.

1.2 Bases

Un diagramme d'états-transitions ne peut être associé qu'à un seul classifieur. Tous les automates à états-finis des diagrammes d'états-transitions d'un système s'exécutent concurrement et peuvent changer d'état de façon indépendante.

1.2.1 Concepts de base

Le diagramme d'états-transitions UML conserve les principes généraux des modèles états – transition : État, Transition, Événement.

Le diagramme d'états-transitions UML ajoute des spécificités : Signal, Transition gardée, Effet (machine à état comportementale).

1.2.2 Notion d'état

État simple (sens strict) :

- ▷ potentiellement *observable au cours de la vie* d'un classifier
- ▷ *durée d'activation non nulle*
- ▷ défini par un invariant
  - ◇ les valeurs d'un ensemble de propriétés du classifier
  - ◇ une situation d'attente
  - ◇ l'exécution d'un comportement

Pseudo-état :

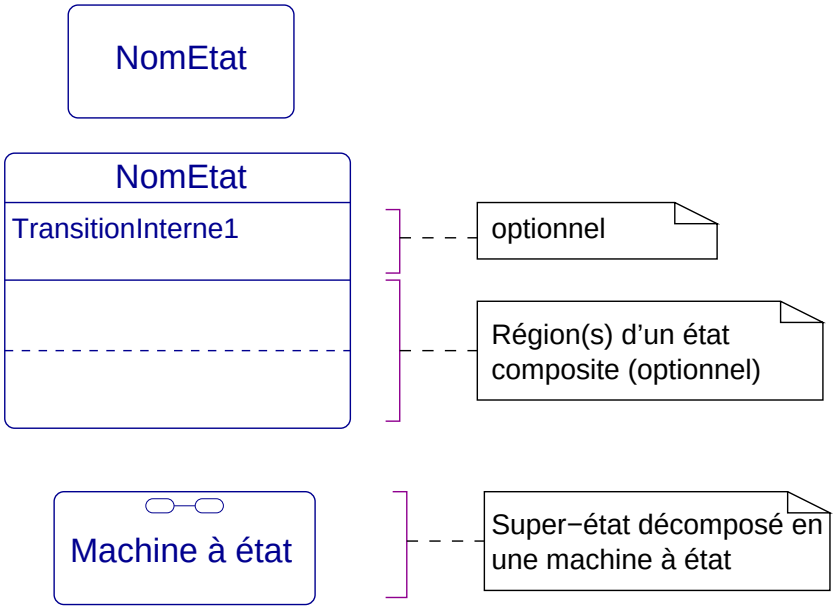
- ▷ non observable, généralement de durée nulle
- ▷ rôle : contrôle des activations des états

État composite

- ▷ peut contenir (envelopper) des sous-états

Notations

Figure 1.2. *Différentes notations*



1.2.3 État initial et état final

Définitions Les états initiaux et finaux sont des pseudo–états caractérisent l'activation (initiale ou terminale) d'une *région* d'un état englobant

État initial (durée nulle) : Lorsqu'un objet est créé, il entre dans l'état initial. L'état initial indique l'activation initiale de la région englobante en ciblant le premier état actif par défaut.

Attention, il existe des restrictions sur la Transition d'un état initial :

- ▷ non déclenchée par un événement,
- ▷ éventuellement gardée (sous réserve...),
- ▷ éventuellement associée à un *effet*

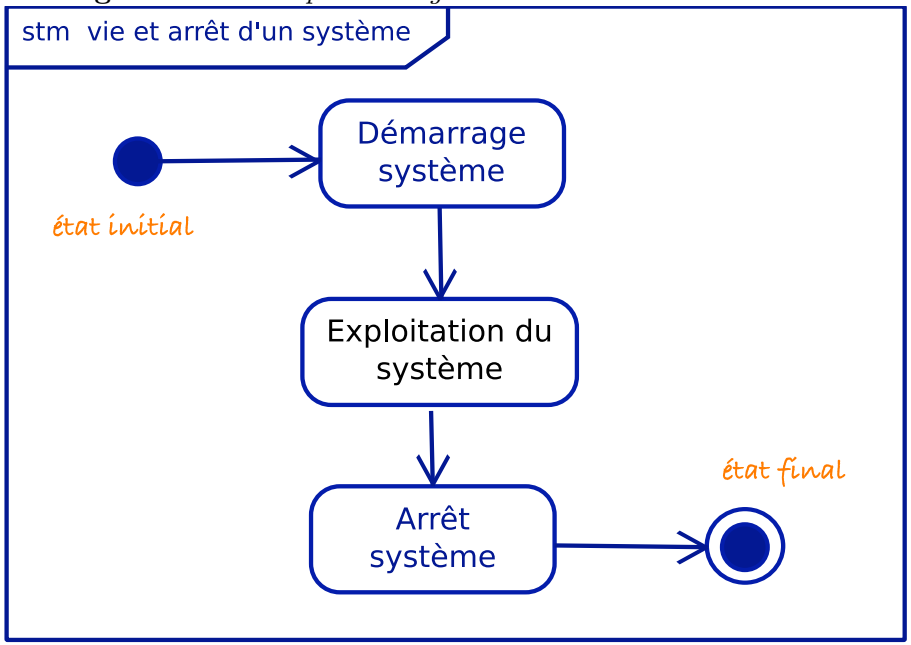
Un état initial est représenté par un rond plein.

État final L'état final indique que l'activation de la région englobante est terminée. Aucun autre état n'est alors activable.

Un état final est représenté par un rond plein entouré d'un cercle.

Exemple

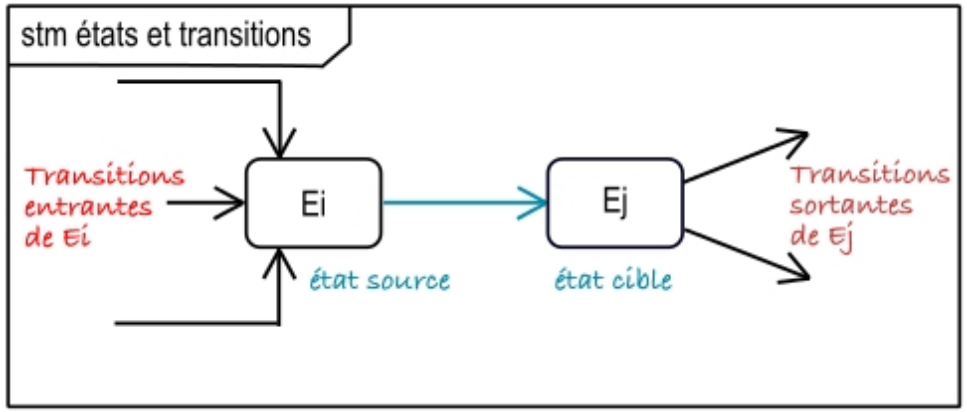
Figure 1.3. *Exemple de diagramme d'états-transitions UML*



1.2.4 Transition

Définition Une transition indique un changement possible d'état du classifier. Elle définit la réponse d'un objet à l'occurence d'un événement. Une transition relie un **état source** à un **état cible**.

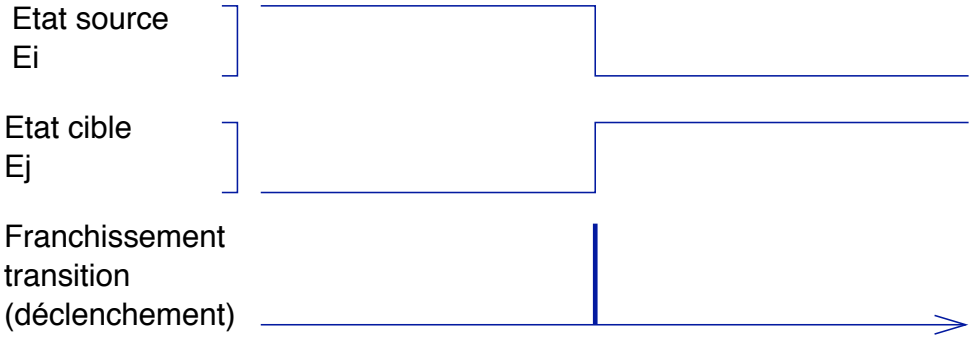
Figure 1.4. *Principe de transition*



Attention : ce diagramme n'est pas syntaxiquement correct. En effet, une transition relie nécessairement 2 états. Cette figure correspond à un extrait du diagramme.

Déclenchement Le changement d'état est instantané (= atomique). Le franchissement (= déclenchement) d'une transition a lieu sur **occurrence d'un événement** . Il peut aussi dépendre d'une **condition de garde**. Le franchissement peut avoir un **effet** sur le classifier = réalisation d'une action.

Figure 1.5. *Franchissement d'une transition*

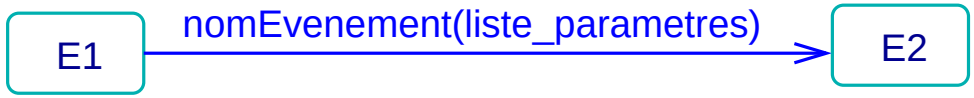


1.2.5 Événement

Définition Un événement est quelque chose qui se produit à un instant donné lors de l'exécution d'un système. Un événement peut véhiculer une information (paramètres) entre un objet émetteur et un objet récepteur. Un événement est associé à une transition (ou plusieurs) l'**occurrence d'un événement** est caractérisée par :

- ▷ une instance d'un type d'événement
- ▷ est instantané (durée nulle)
- ▷ peut provoquer le déclenchement d'une transition (éventuellement plusieurs)

Figure 1.6. *Représentation schématique d'un évenement*



Remarques :

- ▷ une transition peut ne pas avoir d'événement associé ;
- ▷ quand un événement est associé à une transition, on dit aussi qu'elle est synchronisée sur l'événement ;
- ▷ on ne peut pas associer plus d'un événement à une transition ;
- ▷ un même type d'événement peut être associé à plusieurs transitions ;
- ▷ une transition peut à la fois avoir une garde et être syncrhonisée sur un événement.

Types prédéfinis

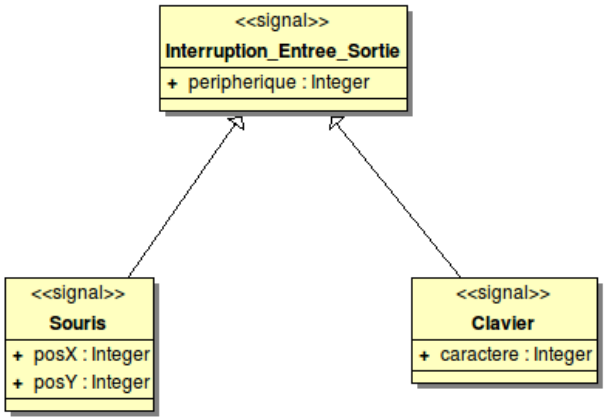
- ▷ call event : `nomOpe(parameter: type, ...)`
  - ◇ réception d'un appel d'invocation d'une opération
  - ◇ le déclenchement de la transition provoque l'exécution de l'opération
  - ◇ l'émetteur retrouve le contrôle lorsque l'opération est exécutée
- ▷ change event : `when(<expression>)`
  - ◇ événement occurrent quand <expression> devient vraie
  - ◇ pas de paramètre
- ▷ signal event : `nomSignal(parameter: type, ...)`
  - ◇ communication asynchrone entre objets

- ◇ (voir la suite)

▷ time event : `after(<time>)`
- ◇ <time> : expression temporelle spécifiant un instant ou durée

Événement de type Signal Un signal est un stéréotype de classifier. Un ≪`signal`≫ peut avoir des attributs et éventuellement des opérations (p. ex. pour sa création). Un ≪`signal`≫ peut être défini comme une spécialisation d'un autre ; la réception d'un signal event provoque le déclenchement d'une transition étiquetée par une de ses généralisations. Il sert à la **communication asynchrone explicite** entre objets Il est produit par une action `send` et peut être destiné à un ou plusieurs objets (explicites).

Figure 1.7. *Exemple de déclaration de signaux*



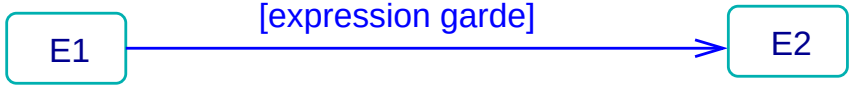
1.2.6 Transition d'achèvement

Une transition d'achèvement est une transition dépourvue d'événement déclencheur explicite. Elle se déclenche à la fin de l'exécution de l'activité de l'état source (y compris les états imbriqués). Elle peut avoir une condition de garde. Celle-ci est évaluée quand l'activité de l'état source s'achève ; elle ne l'est plus ensuite.

1.2.7 Condition de garde d'une transition

La condition de garde d'une transition est une expression booléenne évaluée dans le contexte de l'objet (classifieur) détenteur de la machine à état. C'est une expression logique sur les attributs de l'objet, ainsi que sur les paramètres de l'événement déclencheur. Elle n'est *évaluée que si la transition est éligible* à son déclenchement. En pratique, elle peut correspondre à une opération ayant un paramètre de retour de type `boolean` ; l'opération doit être une requête ({`query`}).

Figure 1.8. *Condition de garde*



La garde est évaluée au moment du franchissement de la transition. Son évaluation ne doit pas avoir d'effet de bord, c'est à dire qu'elle ne doit pas modifier l'état du système, ni produire d'événement.

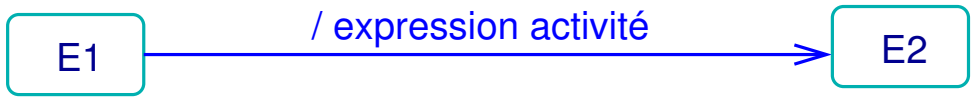
Attention : il ne faut pas confondre une condition de garde et un événement de type change event. Sauriez-vous expliquer la différence ? Dans la pratique, une transition d'achèvement gardée, dans le cas où il n'y a pas d'activité associée à l'état source, ne pose pas de problème : la transition est franchie quand la condition de garde est vraie (cette dernière peut devenir vraie avant ou pendant que le classifieur est dans l'état source, cela ne change rien). Dans le cas où il y a une activité associée à l'état source, il est prudent d'indiquer ce qu'il faut faire si cette activité prend fin alors que la condition de garde n'est pas vraie. Dans le cas contraire il y aurait un blocage potentiel pas très explicite sur le diagramme.

1.2.8 Effet associé à une transition

L'effet associé à une transition est un comportement exécuté lors du franchissement d'une transition. Il peut modifier l'objet détenteur de la machine à état. Il peut utiliser les paramètres de l'événement déclencheur (événement en cours).

Il est lié à une *sémantique run-to-completion* : aucun nouvel événement n'est traité pendant son exécution ; plusieurs actions peuvent être associées à la même transition. En pratique il faut se limiter à des actions « courtes » (voir transitions internes pour les actions « étendues »).

Figure 1.9. *Représentation schématique d'un effet*



Le comportement associé à une transition peut être :

- ▷ une opération primitive comme une instruction d'assignation (modification d'un attribut ou d'un lien de l'objet auquel correspond la machine à état ;
- ▷ l'envoi d'un signal ;
- ▷ l'appel d'une opération ;
- ▷ une liste d'actions (expression des actions séparées par des virgules
- ▷ ...

1.2.9 Exemples

Figure 1.10. *Exemples de transition*

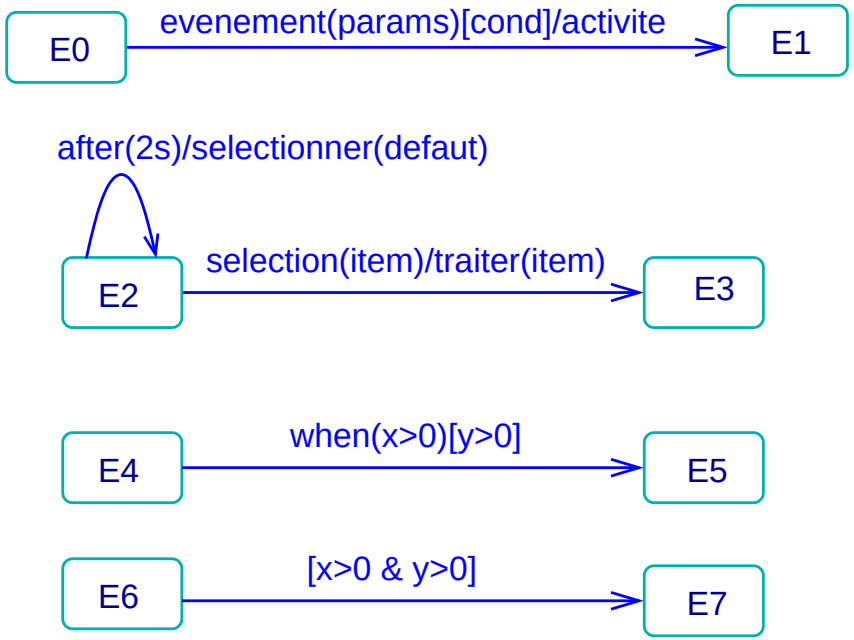
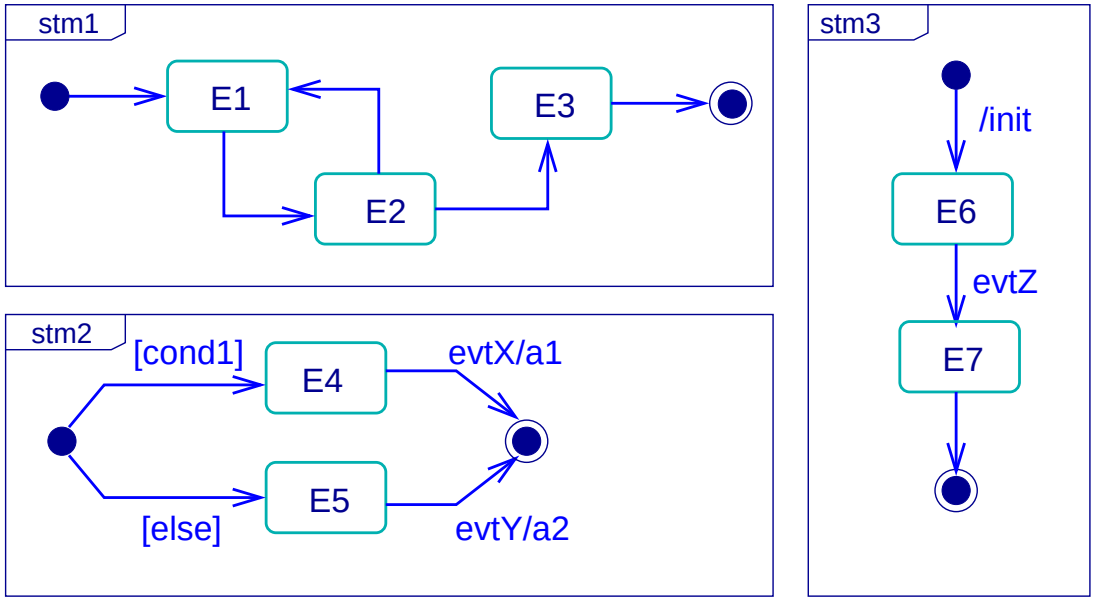


Figure 1.11. *Exemples de diagrammes d'états-transitions*



Note : il a des restrictions sur l'expression d'une transition ayant comme état source un état initial. Il n'y en a pas dans le cas d'une transition ayant pour cible un état final.

1.2.10 Machine à états

Sémantique des machines à état La spécification des séquences d'activation des états permet de définir comment traiter les occurrences d'événements ? Quand ? Dans quel ordre ? Quelle(s) transition(s) déclencher ? 0 ou 1 dans le cas simple ; éventuellement plusieurs dans le cas général. Quels sont les états actifs ? 1 seul dans le cas simple (plusieurs dans le cas des états composites ou hiérarchiques).

La sémantique des machines à état UML diffère des *statemachines* de HAREL ; elle a fortement évolué d'une version d'UML à une autre.

Occurrences d'événements

- ▷ Modèle **asynchrone** : deux événements ne peuvent se produire simultanément et ils sont indépendants

- ▷ Modèle **discret** : on ne traite qu'un événement à la fois
- ▷ Les événements produits sont placés dans un pool d'événements
  - ◇ le pool peut être vide ou contenir plusieurs événements
  - ◇ conceptuellement, l'événement traité est choisi arbitrairement
    une **sémantique opérationnelle** doit spécifier cet ordre
    aléatoire ou *file* (=Paps)
- ▷ run–to–completion : on répercute toutes les conséquences de l'occurrence d'un événement avant d'en traiter un autre

Transitions déclenchées / traitement d'un événement

1. Toutes les transitions étiquetées par cet événement (ou par un événement plus général) et dont l'état source est actif sont **éligibles** (= potentiellement déclenchables)
2. Qu'il y ait ou non une transition éligible, l'événement est consommé (retiré du *pool*)
3. Les gardes associées aux transitions éligibles sont évaluées
   (l'ordre est indifférent) ;
   seules les transitions éligibles dont les conditions de garde sont satisfaites sont déclenchées
4. Les effets associés aux transitions éligibles sont exécutés
5. L'état cible est activé

1.2.11 Exercices

Exo1 On considère la machine à état comportementale suivante d'un classifier `CC`.

1. À sa création, le classifier passe dans l'état `E1` après avoir exécuté l'action `init()`.
2. Étant en `E1`, il passe en `E2` sur occurrence d'un événement `evtX` en exécutant l'action `act1`.
3. Une occurrence de `evtY` en `E2` rend le classifier inactif.
4. Étant en `E1`, si la condition `cond` est vraie, et quand l'événement `evtY` se produit, le classifier passe dans l'état `E3` en exécutant `act2`.
5. Étant en `E3`, `evtY` le refait passer en `E1`.

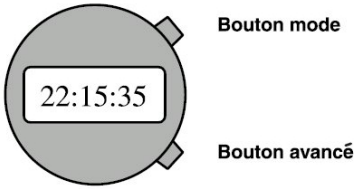
Représentez ce comportement sous forme d'un diagramme états–transitions Uml.

Réveil matin Considérons un réveil-matin simplifié :

- ▷ on peut mettre l'alarme *"on"* ou *"off"*,
- ▷ quand l'heure courante devient égale à l'heure d'alarme, le réveil sonne sans s'arrêter,
- ▷ on peut interrompre la sonnerie.

Question 1 – Dessinez le diagramme d'états correspondant
Question 2 – Complétez le diagramme pour prendre en compte le fait que la sonnerie s'arrête d'elle-même au bout d'un certain temps

Montre à cadran Considérons une montre à cadran numérique simplifiée :



1. Le mode courant est le mode *"affichage"*
2. Quand on appuie une fois sur le bouton mode, la montre passe en ≪ modification heure ≫. Chaque pression sur le bouton avance incrémente l'heure d'une unité.
3. Quand on appuie une nouvelle fois sur le bouton mode, la montre passe en ≪ modification minute ≫. Chaque pression sur le bouton avance incrémente les minutes d'une unité.
4. Quand on appuie une nouvelle fois sur le bouton mode, la montre repasse en mode ≪Affichage ≫.

Question 1 – Dessinez le diagramme d'états correspondant
Question 2 – Ajoutez le comportement suivant : quand on appuie sur le bouton avance plus de deux secondes, les heures (ou les minutes) s'incrémentent rapidement jusqu'à ce qu'il se produise un relâchement dans la pression du bouton. Envisagez plusieurs solutions possibles

1.3 Concepts avancés

1.3.1 Activités et transitions internes

Activités ≪ internes ≫

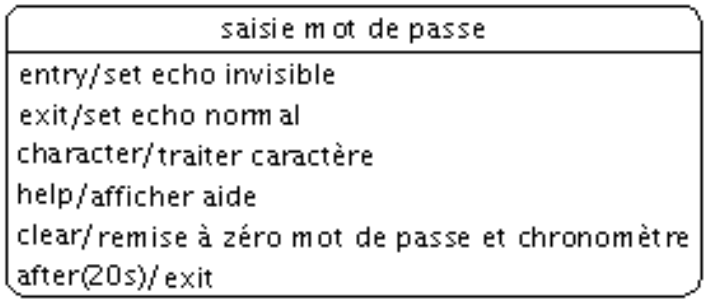
- ▷ Objectif : meilleure encapsulation de la modélisation des comportements en distinguant ce qui dépend de l'environnement (événements – transitions) et de l'état intrinsèque d'un classifier
- ▷ `do / activity` : activité effectuée quand l'état est actif
  elle se termine d'elle-même ou quand l'état n'est plus actif
- ▷ `entry / activity` : activité exécutée lorsque l'état devient actif
  elle est exécutée après les activités des transitions d'entrée et avant les activités `do` et `exit`
- ▷ `exit / activity` : activité exécutée lorsque l'état devient inactif
- ▷ `include / activity` : permet d'invoquer un sous-diagramme d'états-transitions

Les activités `entry` servent souvent à effectuer la configuration nécessaire dans un état. Comme il n'est pas possible de l'éluder, toute action interne à l'état peut supposer que la configuration est effectuée indépendamment de la manière dont on entre dans l'état. De manière analogue, une activité `exit` est une occasion de procéder à un nettoyage.

Transitions ≪ internes ≫

- ▷ transition attachée à un état
  (doit être actif pour qu'elle soit déclenchée)
- ▷ ne provoque pas de changement d'état
  donc pas de réalisation des activités internes `entry` ni `exit`
- ▷ Notation : expression d'une transition mais pas d'arc :
  `nomEVT(liste param)[garde]/activites`

Figure 1.12. *Exemple d'activités/transitions internes*



1.3.2 Alternatives

1.3.3 État composite

1.3.4 Parallélisme

1.3.5 Hiérarchie

1.3.6 Exercices

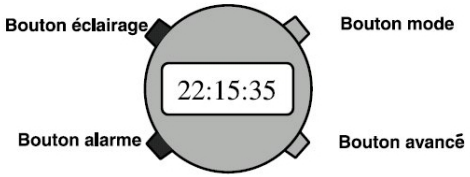
Exo1 On considère la machine à état comportementale suivante d'un classifier CC dont le cycle de vie se décompose en trois grandes phases : Initialisation, Exploitation et Terminaison.

1. À sa création, le classifier est dans l'état Initialisation.
2. Ensuite, sur occurrence de ex, deux comportements sont activés dont les états initiaux sont respectivement E1 et E2. Ces deux comportements indépendants correspondent à la phase d'Exploitation.
3. De l'état E1, sur occurrence de ey on passe en E3, état dans lequel on exécute l'action aa() ; le comportement est ensuite inactif.
4. L'état E2 est un état composite (on ne décrit pas ici ce qui s'y passe) dont on sort si la condition fini est vraie ; le comportement est ensuite inactif.

5. Quand les deux comportements de la phase d'Exploitation sont terminés, l'objet passe dans l'état Terminaison, ce qui termine son cycle de vie.

Représentez ce comportement sous forme d'un diagramme états–transitions UML.

Régions concurentes Reprenons l'exemple de montre à cadran numérique, et ajoutons maintenant à cette dernière deux autres boutons :



- ▷ un bouton éclairage ; en le pressant, on éclaire le cadran de la montre, jusqu'à ce qu'on le relâche ;
- ▷ un bouton alarme, qui ajoute à la montre digitale une fonctionnalité classique d'alarme, comme cela a été décrit lors du premier exercice de ce chapitre (réveille-matin).

Question – Dessinez le diagramme d'états complet incluant tous les comportements de la montre.

Grapher On s'intéresse ici à la modélisation du comportement du grapher quand l'utilisateur interagit avec cet objet graphique au clavier ou avec la souris. On suppose que le système de gestion de l'interface utilisateur produit des signaux du type `KB(n: string)` lorsque l'utilisateur presse une touche nommée `''n''` et `Mouse` s'il utilise la souris et que le grapher peut y réagir.

1. Au départ le grapher est vide : pas de surface affichée.
2. Si l'on fait appel à l'opération `display(in s: Surface)`, alors cela a pour effet d'exécuter l'action `draw(in s: Surface)`. Une surface est maintenant affichée.
3. Si une surface est affichée, presser la touche "+" déclenche `zoomIn()` et la touche "-" `zoomOut()`. La touche "space" supprime la surface du grapher.

4. Toujours quand une surface est affichée, presser la touche “z” fait passer le grapher dans un mode ou l'on affiche un plan de coupe sur l'axe Oz. Lorsqu'on active ce mode, le plan de coupe est affiché, ainsi que sa position selon Oz. Si l'on bouge la souris, cela déplace le plan (`movePlan()`) ; si l'on appuie sur la touche “space”, cela masque la position ; une nouvelle pression sur la touche “space” rend la position de nouveau visible. Dans tous les cas, presser la touche “z” fait sortir du mode plan de coupe.

Représentez ce comportement sour forme d'un modèle état–transition d'UML.

Photos : Prise d'une photo On s'intéresse ici à ce qui se passe juste avant, pendant et après la prise d'une photo avec un appareil numérique. Cela concerne donc la manipulation du déclencheur de l'appareil. Ce bouton est soit relâché, soit enfoncé à mi-course, soit enfoncé à fond.

1. Au départ, le déclencheur est relâché.
2. Une fois que l'on a enfoncé le déclencheur à mi-course, l'appareil effectue la mise au point.
3. À partir de là, si l'on enfonce le déclencheur à fond, cela déclenche la prise de la photo.
4. Notons que si l'on enfonce directement le déclencheur à fond, l'appareil prend une photo, sans changer de réglage.
5. Quand la prise de vue est achevée, l'appareil enregistre la photo sur le support de stockage (cette opération peut prendre un certain temps). La prise de vue est ensuite considérée comme terminée.

Modélisez ce comportement sous forme d'un diagramme états–transitions UML.

Photos : Réglage de l'exposition On modélise ici les différents modes de réglage de l'exposition sur un appareil photo : mode auto-focus (`AF`), mode manuel avec priorité à l'ouverture (`PO`) et mode manuel avec priorité à la vitesse (`PV`).

Par défaut, l'appareil est en mode `AF`. Le bouton `man` passe dans un des modes `Manuel`, par défaut en mode `PV` ou dans le dernier mode manuel sélectionné. On peut passer du mode `PO` au mode `PV`, et vice versa, à l'aide du bouton `man`. En mode `PO` ou `PV`, le bouton `af` refait passer en mode auto focus. En mode `AF`, on peut aussi activer le mode 'macro' à l'aide du bouton du

même nom, ce qui fait passer en mode `PO`.

> Modélisez la gestion de ces modes sous forme d'un diagramme états–transitions UML.

Photos : Utilisation du flash L'objectif est de modéliser les différentes possibilités de réglage du flash d'un appareil photo, combiné avec le mode d'attenuation de l'effet « yeux rouges ».

Il y a deux grands modes de fonctionnement : sans flash ou avec flash. L'activation du mode « yeux rouges » n'est possible que quand le flash est actif. Au démarrage, l'appareil est réglé en mode 'flash Auto' et 'attenuation yeux rouges', noté `Attenuation YR`. Lorsque le flash est actif, on peut activer et désactiver le mode `Attenuation YR`. On passe de l'un à l'autre à l'aide du bouton yr. Trois modes de fonctionnement du flash sont possibles : 'flash auto', 'flash forcé' et 'flash doux'. On passe de l'un à l'autre, et dans cet ordre, par appui sur le bouton flash. On passe dans le mode 'sans flash' avec le bouton no flash. Dans ce mode, on peut réactiver le flash (bouton flash), ce qui fait passer en mode 'flash forcé' et 'attenuation yeux rouges'.

> Modélisez la gestion de ces modes sous forme d'un diagramme états–transitions UML.

La vie des animaux Le comportement d'un lion est le suivant, méthode `live()`.

1. Par défaut, un lion se repose.
2. Quand il a faim (`isHungry():boolean {query}`), il chasse.
3. S'il est épuisé (`isExhausted():boolean {query}`), il se repose à nouveau.
4. Quand un lion chasse et qu'il capture un proie, il la mange (`eatAnimal()`) et il se repose ensuite.

> Représentez ce comportement de la classe Lion par un diagramme états–transitions.

On va maintenant raffiner le comportement de chasse des lions.

1. Le lion regarde s'il y a une proie dans son voisinage.
2. S'il voit une proie, il se déplace dans la case correspondante, sinon, il se déplace aléatoirement.
3. Si la proie est toujours dans la case, il la capture ; si la proie n'y est plus, le lion regarde à nouveau dans le voisinage.

Modélisez ce comportement : identifiez les opérations correspondantes de la classe `Lion` et l'enchaînement possible de leur réalisation sous forme d'un diagramme états–transitions.

1.4 Mise en œuvre

2 Diagramme d'activités

Sommaire

2.1 Activités : introduction

2.2 Activités : concepts de base

2.2.1 Activité et action

2.3 Flot de contrôle et de données

2.4 Activités : concepts avancés

2.4.1 Nœuds de décision et de fusion

2.5 Débranchement et branchement

2.6 Autres éléments de contrôle

2.7 Structuration des modèles

3 Génération de code

Sommaire

3.1 Diagramme de classes

3.1.1 Principe

UML est indépendant des langages de programmation et des technologies. Comment implémenter (=coder) les modèles ? Certains éléments de modélisation sont communs à UML et aux LPO :

- ▷ Ont-ils la même sémantique ?
- ▷ Sinon, comment assurer la traduction de l'un dans l'autre ?
- ▷ Existe-t'il une et une seule traduction possible ?
- ▷ La traduction est-elle « réversible » ?

Il faut définir des *profiles* adaptés.

Définition 3.1. *LPO : langage de programmation (orienté) objet.*
Exemples de langage : C++, java, C#, Smalltalk, Objective C...

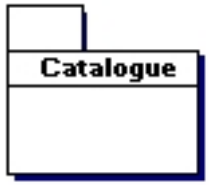
3.1.2 Package

- ▷ Soit un `Package` de nom `"pkgName"`
- ▷ Dans le répertoire du contexte courant, création d'un répertoire `pkgName` (**Java** et **C++**)
- ▷ **Java** (dans chaque fichier correspondant aux classes du package) : `package pkgName;`
- ▷ **C++** :
  - ◇ Un répertoire `pkgName` dans l'arborescence `include`
  - ◇ Un répertoire `pkgName` dans l'arborescence `src`
  - ◇ Une bibliothèque dans `lib` (p. ex. `libpkgName.so`)
  - ◇ `namespace pkgName {...};`

Dépendance

- ▷ **Java** : `import pkgName.*;`
- ▷ **C++** : `using namespace pkgName;`

Convertion

UML	Java
	`package Catalogue` `...`
	C++

	`namespace Catalogue`
	{
	`...`
	}

3.1.3 Classes

Soit une `Class CName` du package `pkgName`

- ▷ Un fichier par classe
  (sauf pour les classes contenues dans une autre classe)
- ▷ **Java** : fichier CName.java dans le répertoire du package
  package pkgName ;
  public class CName {...}
- ▷ **C++** :
  - ◇ CName.h dans l'arborescence `include` (rép. du `pkgName`)
  - ◇ CName.cpp dans l'arborescence `src` (rép. du `pkgName`)
  - ◇ fichier CName.o (bibliothèque libPkgName.so)
  - ◇ namespace pkgName {
    class CName {...} ;
    }

Cas particuliers

- ▷ Classe abstraite
  - ◇ Java :
    public abstract class AA { ... }
  - ◇ C++ :
    au moins une opération abstraite

- ▷ Interface :
  - ◇ Java :
    public interface IJ { ... }
  - ◇ C++ :
    classe dont toutes les opérations sont abstraites

En C++, une classe *interface* n'a aucune variable membre non statique, ni de constructeur explicite et ses fonctions membres sont toutes virtuelles pures ; comme toujours, le destructeur est virtuel.

3.1.4 Opération

- ▷ *Operation vs Method*
  - ◇ Operation : spécification du service (déclaration)
  - ◇ Method : définition de l'opération
  - ◇ Une Operation sans Method associée est *abstraite*
- ▷ *Paramètres* :
  - ◇ Type : voir traduction des types de base
  - ◇ Direction : `in, out, inout, return`

3.1.5 Types

Java

▷ ≪ type ≫ integer – TaggedValue {size}, {wrapped}

	—	{wrapped}
—	int	Integer
{size,long}	long	Long
{size,short}	short	Short
{size,byte}	byte	Byte

▷ ≪ type ≫ real – TaggedValue {size}, {wrapped}

	—	{wrapped}
—	float	Float
{size,long}	double	Double

Remarque : il s'agit d'une possibilité de traduction, les TaggedValue {size} et {wrapped} ne font pas partie de la norme UML.

▷ ≪ type ≫ boolean – TaggedValue {wrapped}

—	{wrapped}
boolean	Boolean

▷ ≪ type ≫ string

String

▷ ≪ type ≫ char – TaggedValue {wrapped}

—	{wrapped}
char	Character

C++

▷ ≪ type ≫ integer – TaggedValue {size}, {unsigned}

	—	{unsigned}
—	int	unsigned int
{size,long}	long	unsigned long
{size,extraLong}	long long	unsigned long long
{size,short}	short	unsigned short

▷ ≪ type ≫ real – TaggedValue {size}

—	float
{size,long}	double
{size,extraLong}	long double

▷ ≪ type ≫ boolean :

```
bool
```

▷ ≪ type ≫ string :

```
string
#include <string>
```

▷ ≪ type ≫ char :

```
char
```

3.1.6 Attributs

Principes généraux de la transformation

- ▷ *implémentation non accessible* en dehors de la classe
- ▷ *acccessibilité* (lecture, modification) assurée par des méthodes dont l'accessibilité correspond à la *visibilité du modèle de classe*

Les principes de la transformation sont communs à tous les langages objets typés et supportant le principe d'encapsulation.

C++ Exemple d'un attribut `attrName: AttrType` protégé

```
class CName {
  protected:
    inline AttrType getAttrName(void) const;
    inline void setAttrName(AttrType value);
    inline virtual void _invariant(void);

  private:
    AttrType _attrName;
};


inline AttrType CName::getAttrName(void) const {
  return _attrName;
}
inline void CName::setAttrName(AttrType value) }
  _attrName = value;
  _invariant();
}


inline void CName::_invariant(void) {
  // Begin
  // End
}
```

Java Exemple de la classe `Plant`

`+quantity: real = 0`
≪invariant≫ {quantity ∈ [0, maxQuantity]}

▷ *Implémentation privée* (cf. règle de codage) :

```
private double quantity;
```

▷ *Accesseur public* (cf. modèle) :

```
public double getQuantity() {
  return quantity;
}
```

▷ *Modificateur* : visibilité et maintien de l'invariant

```
public void setQuantity(double quantity) {
  quantity = Math.max(quantity, 0.0);
  quantity = Math.min(quantity, maxQuantity);
  this.quantity = quantity;
}
```

```
+growthRate: real = 0.01 {readOnly}
```

```
public Plant(..., double growthRate) {
  super(...);
  this.growthRate = growthRate;
}

public Plant(...) {
  this(..., 0.01 /* initial value of growthRate */);
}

private final double growthRate;

public double getGrowthRate() { return growthRate; }
```

+maxQuantity: real = 1.0 {readOnly}

```
private static double maxQuantity = 1;

public static double getMaxQuantity() {
  return maxQuantity;
}
```

birthdate: real = Clock.time {readOnly}
+/age: real = Clock.time - birthdate

```
private double birthdate;

protected double getBirthdate() {
  return birthday;
}


// — Derived attribute 'age'
public double getAge() {
  return Clock.getTime() - this.getBirthdate();
}
```

Java : attribut à multiples valeurs : Exemple de la classe SquareCell

`+position: real[2]`

▷ Traduction de `real` sous forme d'un *wrapper* :

```
private Vector<Double> position = new Vector<Double>(2);
public SquareCell(double x, double y) {
   position.set(0, x);
   position.set(1, y);
}
```

▷ Traduction de `real` sous forme d'un **type de base** :

```
private double[] position = new double[2];
public SquareCell(double x, double y) {
   position[0] = x;
   position[1] = y;
}
```

Masquage de l'implémentation

▷ *wrapper* :

```
  public double getPosition(int index) {
    return position.elementAt(index);
}
  public void setPosition(int index, double value) {
    position.set(index, value);
}
```

▷ type de base :

```
  public double getPosition(int index) {
    return position[index];
}
   void setPosition(int index, double value) {
    position[index] = value;
}
```

3.1.7 Instanciation

Instanciation d'une classe UML : constructeurs

- ▷ *Constructeurs implicites* :
  - ◇ nom de la méthode : nom de la classe
  - ◇ valeurs initiales des attributs
  - ◇ construction des conteneurs pour l'implémentation des attributs à valeurs multiples et des associations de multiplicité > 1
  - ◇ action(s) des états initiaux (cf. modèle état–transition)
  - ◇ Cohérence des constructeurs
- ▷ *Constructeurs explicites* :
  `Operation` stéréotypée ≪`create`≫

Java exemple de la classe Clock : constructeurs

```java
package simLife;

public class Clock {

  public Clock(double initialTime, double deltaT,  double timeMax) {
    this.time = initialTime;
    this.deltaT = deltaT;
    this.timeMax = timeMax;
    // ...
  }

  public Clock(double initialTime, double deltaT) {
    this(initialTime, deltaT, Double.MAX_VALUE);
  }

  public Clock() {
    this(0.0, 1.0, Double.MAX_VALUE);
  }
```

C++ exemple de la classe CName : constructeurs

Soit `CName` ayant deux attributs :
`attr1: integer = 0`
`attr2: string = ""`

```
// File: CName.h
...
class CName {
  public:
    CName(void);
    CName(const CName & instanceOfCName);
    const CName & operator = (const CName & instanceOfCName);
    virtual ~ CName(void);
    ...
  protected:
    virtual void _copy(void);
    ...
};
```

```
// CName.cpp


CName::CName(void): attr1(0), attr2("") {
  // Begin
  // End
}

CName:: CName(const CName & instanceOfCName) {
  _copy(instanceOfCName);
}

const CName & CName::operator = (const CName & instanceOfCName) {
  if (this != & instanceOfCName) _copy(instanceOfCName);
  return * this;
}

CName::~ CName(void) {
  // Begin
  // End
}

void CName::_copy(const CName & instanceOfCName) {
  this- >setAttr1(instanceOfCName.getAttr1());
  this- >setAttr2(instanceOfCName.getAttr2());
}
```

3.1.8 Association

Java Association unidirectionnelle en Java :Exemple de CName1 vers CName2

▷ `Multiplicity = 0..1` (p. ex. visibilité `public`)

```
class CName1 {

// — Implémentation cachée de l'objet en association
  private CName2 roleName;



// — Access public à l'objet en association
  public CName2 getRoleName() {
    return this.roleName;
  }


// — Modification possible de la référence de l'objet en association
  public void setRoleName(CName2 instance) {
    if (!this.getRoleName().equals(instance))
      this.roleName = instance;
  }
}
```

▷ `Multiplicity = 0..*` ou `1..*` (p. ex. visibilité `public`)

```
class CName1 {
  // — Implémentation cachée de la collection (= ensemble)
    private Set<CName2> roleName = new HashSet<CName2>();


  // — Access public à l'ensemble d'objets en association

    public Set<CName2> getRoleName() {
     return new Set<Cname2>(roleName); }
    public void setRoleName(Set<CName2> setOfCName2) { ... }


  // — Modification de l'ensemble d'objets en association

    public boolean addToRoleName(CName2 instance) { ... }
    public boolean removeFromRoleName(CName2 instance) { ... }


  // — Informations sur l'ensemble d'objets en association

    public boolean hasInRoleName(CName2 instance) { ... }
    public int cardOfRoleName() { ... }
}
```

C++

▷ Multiplicity = 0..1

```
class CName1 {
  public:
    inline virtual const CName2 * getRoleName(void) const;
    inline virtual void setRoleName(CName2 * instance);
  private:
    CName2 * _roleName;
};


inline const CName2 * CName1::getRoleName(void) const {
  return _roleName;
}


inline void CName1::setRoleName(CName2 * instance) {
  if (getRoleName() != instance) {
    _roleName = instance;     _invariant();
  }
}
```

Multiplicity = 0..* ou 1..*

```
class CName1 {
  public:


// — Global access to the set of instances 'roleName'
    inline virtual const set < CName2 * > & getRoleName(void) const;

    inline virtual void setRoleName(const set < CName2 * > & setOfInstance);


// — Add/remove one instance in/from 'roleName'
    inline virtual bool addToRoleName(CName2 & instance);

    inline virtual bool removeFromRoleName(CName2 & instance);



// — Get information about the set 'roleName'
    inline virtual bool hasInRoleName(const CName2 & instance) const ;

    inline virtual unsigned int cardOfRoleName(void) const;



// — Hidden implemention of 'roleName'
  private:
    set < CName2 * > _roleName;
};
```

Java Association bidirectionnelle en Java : Gestion de l'intégrité du référencement réciproque

exemple : `Animal` – `Place`

```
private Place location;

public Place getLocation() { return location; }

public void setLocation(Place location) {
  if (location != null)
    if (this.location != null && this.location != location)
      this.location.removeFromLocalPopulation(this);
    if (this.location != location) {
      this.location = location;
      location.addToLocalPopulation(this);
     }
   }
}
```

```
class Place {

   private Set<Animal> localPopulation = new HashSet<Animal>();

   public void addToLocalPopulation(Animal animal) {
    if (!this.localPopulation.contains(animal)) {
      this.localPopulation.add(animal);
      animal.setLocation(this);
    }
  }

}
```

Classe–Association en Java : Exemple de la relation prédateurs – proies (classe Species)

```
class Species {
   private Set<FoodLink> preys = new HashSet<FoodLink>();
  ...
   private Set<FoodLink> predators = new HashSet<FoodLink>();
  ...
}


class FoodLink {
  private Species prey;
  private Species predator;
  private double preference;
  ...
}
```

```
public boolean hasInPreys(Species prey) {
  FoodLink fl = new FoodLink(this, prey, 0);
  Iterator<FoodLink> iter = preys.iterator();
  boolean found = false;
  while (iter.hasNext() &&!found) found = iter.next().equals(fl);
  return found;
}



public boolean addInPreys(Species prey, double preference) {
  // check that the species is not already referenced as a prey
  FoodLink fl = new FoodLink(this, prey, preference);
  Iterator<FoodLink> iter = preys.iterator();
  boolean found = false, added = false;
  while (iter.hasNext() &&!found) found = iter.next().equals(fl);
  if (!found) added = preys.add(fl) && prey.predators.add(fl);
  return added;
}
```

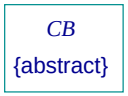
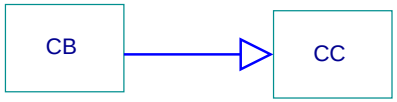
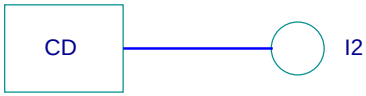
```
public boolean hasInPredators(Species predator) {
  FoodLink fl = new FoodLink(predator, this, 0);
  Iterator<FoodLink> iter = predators.iterator();
  boolean found = false;
  while (iter.hasNext() &&!found)
   found = iter.next().equals(fl);
  return found;
}
```

```
public class FoodLink {
public FoodLink(Species predator, Species prey) {
  this.prey = prey;
  this.predator = predator;
}

public FoodLink(Species predator, Species prey, double preference) {
  this.prey = prey;
  this.predator = predator;
  this.preference = preference;
}

public boolean equals(Object o) {
  if (!(o instanceof FoodLink)) return false;
  FoodLink f = (FoodLink)o;
  return f.prey == this.prey && f.predator == this.predator;
}
```

Association qualifiée en Java : Exemple : `Savanna` – `Species`

```
private static Map<String, Species> species = new HashMap<String, Species>();


public static Species getSpecies(String name) {
  return Savanna.species.get(name);
}

public static void addToSpecies(String name, Species species) {
  Savanna.species.put(name, species);
}

public static void removeFromSpecies(String name) {
  Savanna.species.remove(name);
}

public static boolean hasInSpecies(String name) {
  return Savanna.species.containsKey(name);
}

public static int numberOfSpecies() { return Savanna.species.size();}
```

Classes, interface, généralisation, implémentation

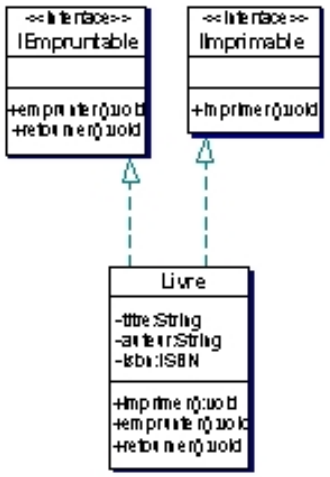
	java	public class CA { /* */ }
	C++	class CA { /* */ };
	java	public abstract class CA { /* */ }
	C++	class CB { /* */ };
	java	public interface I1 { /* */ }
	C++	class I1 { /* */ };
	java	public class CB extends CC { /* */ }
	C++	class CB: public CC { /* */ };
	java	public class CD implements I2 { /* */ }
	C++	class CD: public I2 { /* */ };

Les concepts de classe abstraite et d'interface n'existent pas en tant que tel en C++ ; cependant il est possible de traduire ces notions en classes C++ ayant des propriétés équivalentes.

3.1.9 Autres relations

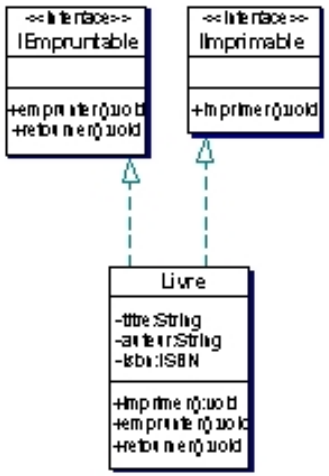
Réalisation en Java

UML	Java

	`public class Livre implements IImprimable, IEmpruntable {`
	`private String titre;`
	`private String auteur;`
	`private ISBN isbn;`
	`public void Imprimer(){`
	`...`
	`}`
	`public void Emprunter(){`
	`...`
	`}`
	`public void Retourner(){`
	`...`
	`}`
	`}`

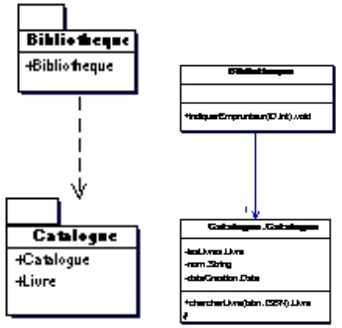
Réalisation en C++

UML	C++

	`public class Livre :`
	`IImprimable, IEmpruntable {`
	`private string titre;`
	`private string auteur;`
	`private ISBN isbn;`
	`public void Imprimer(){`
	`...`
	`}`
	`public void Emprunter(){`
	`...`
	`}`
	`public void Retourner(){`
	`...`
	`}`
	`}`

Dépendance

UML	Java

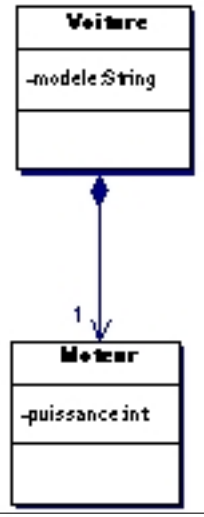
	`package Bibliotheque;`
	`import catalogue;`
	`public class Bibliotheque {`
	`private Catalogue leCatalogue;`
	`...`
	`}`
	`}`

	C++

	`namespace Bibliotheque {`
	`using Catalogue;`
	`public class Bibliotheque {`
	`private Catalogue leCatalogue;`
	`...`
	`}`
	`}`

Agrégation

UML



Java

```
public class Voiture {
 private String modele;
 private Moteur moteur;
 private static class Moteur {
 private int puissance;
 }
 ... }
```

C++

```
public class Voiture {
 private string modele;
 private Moteur moteur;
 private class Moteur {
 private int puissance;
 }
 ... }
```

3.1.10 Exercices

Java => diagramme de classes On considère le code JAVA ci-dessous et pages suivantes.

Traduire ce code en un diagramme de classes UML.

```
// File: AA.java ------------------------------------------------------
package pkgA;

abstract public class AA {

       public abstract float ope1();
}
// ------------------------------------------------------------------
```

```
// File: I1.java ------------------------------------------------------
package pkgA;

public interface I1
{
       boolean service1();
}
// ------------------------------------------------------------------
```

```
// File: BB.java -----------------------------------------------------
package pkgA;

public class BB implements pkgA.I1
{
       private int attr = 1;
       protected int getAttr() {
             return this.attr;
       }
       protected void setAttr(int value) {
             this.attr = value;
       }

       private pkgA.CC theCC;
       protected pkgA.CC getTheCC() {
             return this.theCC;
       }
             protected int cardTheCC() {
             if ( this.theCC == null ) return 0;
             else return 1;
       }

       public boolean service1() {
             // ...
       }
} // ---------------------------------------------------------------------
```

```
// File: App.java ----------------------------------------------------
package pkgA;

public class App
{
       public static void main(final String[] args) {
             CC objCC = new CC();
       }
}
// ---------------------------------------------------------------------
```

```
// File: ACC.java ----------------------------------------------------
package pkgA;

import java.util.*;

public class CC extends pkgA.AA
{
       private Vector someBB = new Vector();


       public pkgA.BB getSomeBB(int i) {
             return (pkgA.BB)this.someBB.elementAt(i);
       }
       public void setSomeBB(int i, pkgA.BB value) {
             this.someBB.setElementAt(value, i);
       }
       public void appendSomeBB(pkgA.BB value) {
             this.someBB.addElement(value);
       }
       public void eraseSomeBB(pkgA.BB value) {
             this.someBB.removeElement(value);
       }
       public void eraseSomeBB(int i) {
             this.someBB.removeElementAt(i);
       }
       public int cardSomeBB() {
             return this.someBB.size();
       }

       public float ope1() {
             float result = 1.0;
       return result;
       }

       public int ope2(final float Parameter, pkgA.BB objBB) {
             int result = 1;
             objBB.service1();
             return result;
       }
}
// ------------------------------------------------------------------
```

3.2 Diagramme de séquences

3.3 Diagramme d'états-transitions

3.4 Diagramme d'activités

Typologie des actions

- ▷ Création – destruction d'objet
- ▷ Lecture – Modification de la valeur d'un attribut
- ▷ Mise en relation d'objets | suppression de la relation
- ▷ Appel – exécution d'une opération
- ▷ Exécution d'une fonction modifiant la valeur d'une variable
- ▷ Communication : émission – réception de message

Labo

4 Le modèle états-transitions : concepts de base

Sommaire

4.1 Objectif

Ces exercices portent sur le modèle états–transitions du langage UML :
▷ états et des transitions ;
▷ déclencheurs des transitions : cas simples

4.2 Comportement de l'interface d'un téléphone

On considère le comportement simplifié de l'interface d'un téléphone mobile et on souhaite la modéliser en UML. On s'intéresse ici à la fonctionalité 'appel d'un correspondant'.

Q 4.1. Quel modèle d'UML permet de représenter les fonctionnalités ? Représentez la fonctionnalité p'appel correspondant' par un diagramme.

Description de la procédure d'appel.

1. Lorsqu'on active le téléphone, un écran d'accueil s'affiche.
2. Si l'on appuie sur la touche Fin, l'appareil se met en veille et aucune autre interaction n'est alors possible sans une réactivation de celui-ci.
3. Depuis l'écran d'accueil, la touche 'Composer' permet d'accéder à la fonctionnalité d'appel. Un écran d'invitation à composer un numéro s'affiche.
4. L'appui sur la touche 'Fin' ou sur la touche 'Accueil' ramène à l'écran d'accueil.

Q 4.2. Représentez ce comportement avec un diagramme états–transitions d'UML.

1. L'appui sur une touche numérique provoque l'affichage du chiffre correspondant à l'écran.
2. La touche 'Valider' provoque l'appel du numéro, sauf si aucun chiffre n'a été saisi.
3. Lors de l'appel, quand la ligne est détectée comme occupée, l'appareil affiche le message 'occupé' et, après 1 seconde, l'écran de composition s'affiche à nouveau.
4. Lors de l'appel, on peut abandonner l'opération en appuyant sur la touche 'Fin'.
5. Lors de l'appel, si le correspondant décroche, la communication est établie.
6. En cours de la conversation, si le correspondant raccroche, on affiche un message et on revient à l'écran d'accueil.
7. En cours de conversation, la touche 'Fin' interrompt la communication et l'écran d'accueil s'affiche à nouveau.

Q 4.3. Représentez ce comportement à l'aide d'un diagramme états–transitions UML.

Q 4.4. Identifiez-vous des ambiguïtés dans la description textuelle du comportement telle qu'elle vous est donnée ? Votre modélisation permet-elle de les lever ?

4.3 Utilisation d'un éditeur de document

L'étude porte sur une partie du comportement d'un éditeur de document tel qu'un traitement de texte, un éditeur graphique, un logiciel de retouche d'image...

1. Au lancement du logiciel, l'espace de travail est vide.
2. La combinaison de touches `ctrl-n` permet de créer un nouveau document et de l'éditer.
3. `ctrl-f` permet d'ouvrir un document existant dans un fichier : cela ouvre la fenêtre de sélection de fichier. Si l'utilisateur sélectionne un fichier, le document est ouvert et est éditable. Si aucun fichier n'est sélectionné, on revient à l'état initial.

4. En mode "édition", `ctrl-s` permet d'enregistrer le document dans un fichier : s'il s'agit d'un nouveau document, l'utilisateur est invité à saisir un nom de fichier (fenêtre de saisie de fichier), si le document correspond à un fichier ouvert précédemment, `ctrl-s` l'enregistre dans ce même fichier. Dans les deux cas, on peut à nouveau éditer le fichier.
5. Pour enregistrer le document dans un nouveau fichier, la combinaison de touches est `shift-ctrl-s`.

Q 4.1. Quel type de machine à états est adapté à la modélisation de ce protocole opératoire ?

Q 4.2. Modélisez ce mode opératoire sous forme d'un diagramme états–transitions.

5 Le modèle états-transitions : concepts avancés

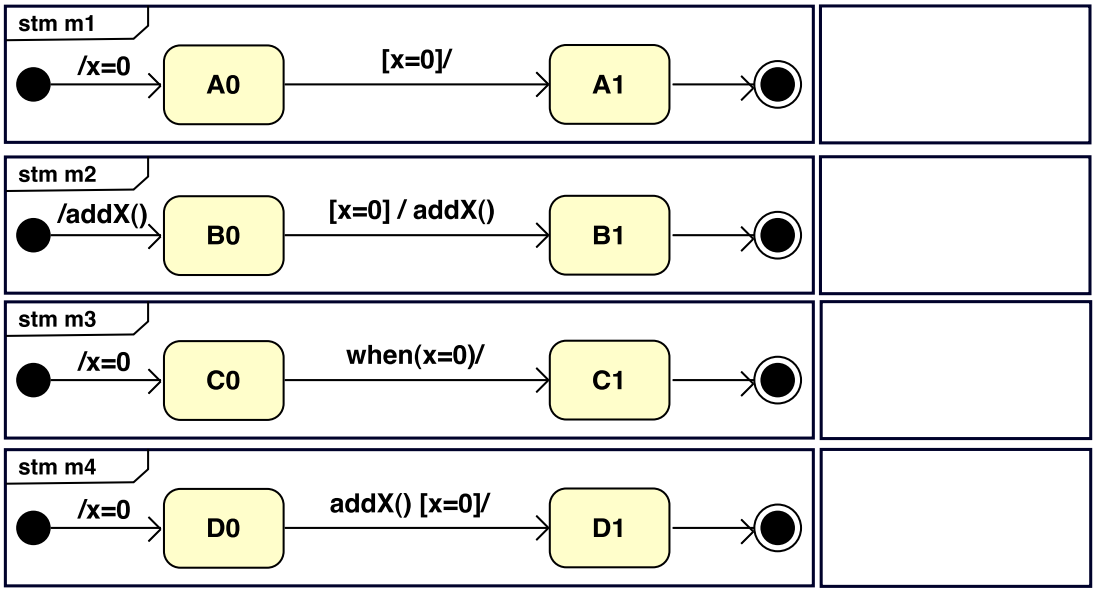
Sommaire

5.1 Objectif

Ces exercices portent sur le modèle états–transitions du langage UML : modélisation comportementale ; interprétation d'une machine à états ; transitions et activités internes ; états hérarchiques (cas simples).

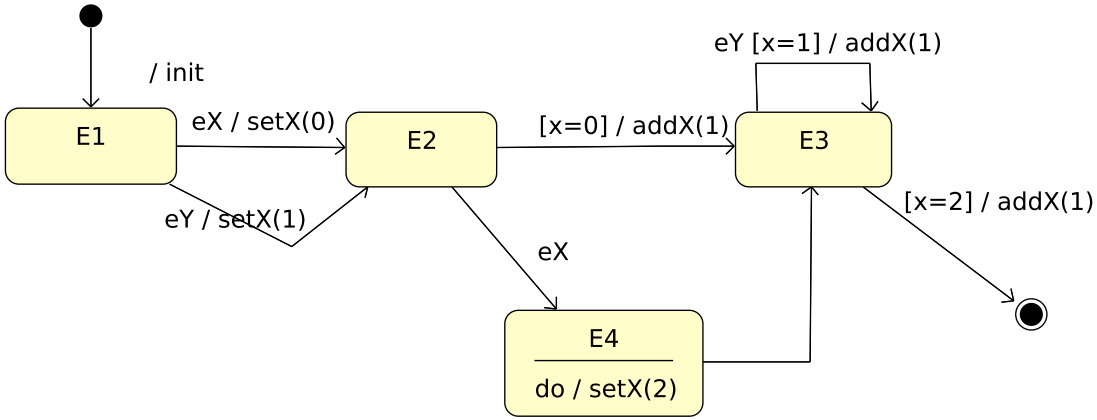
5.2 Sémantique des machines à état

On s'intéresse ici au comportement des objets d'une classe `CC` qui a un attribut `x: integer = 0` et deux opérations `setX(in v:integer)` et `addX(in v: integer = 1)`. La première affecte la valeur `v` à l'attribut `x` ; la seconde incrémente l'attribut `x` de `v`.



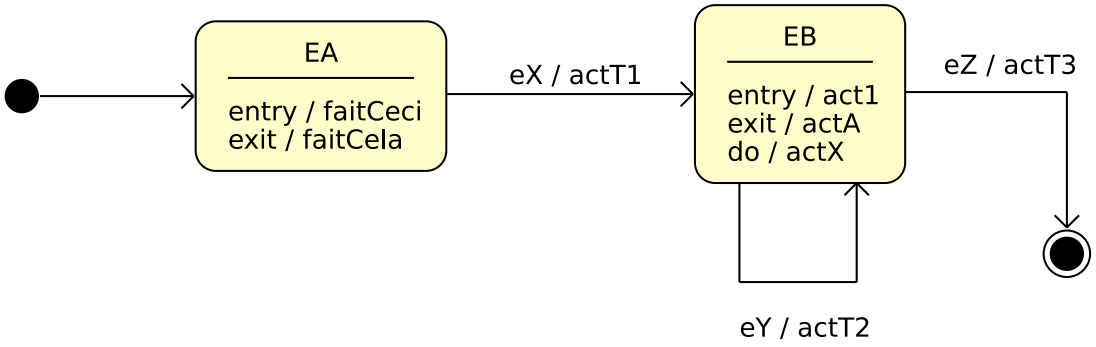
Q 5.1. Quelles sont les transitions franchissables ?

On associe maintenant la machine à états de la figure ci-dessous à la classe `CC`.



Q 5.2. Quelles transitions sont franchies si la séquence d'événements `eX; eY; eX` se produit ? Quelles sont les actions qui sont effectuées et dans quel ordre ? Même question avec la séquence `eY; eX; eY`.

On associe maintenant la machine à état de la figure ci-dessous à la classe `CC`.



Q 5.3. Quelles activités sont effectuées à l'activation de la machine à état ? Que se passe-t'il ensuite sur occurrence d'un événement `eX` ? Que se passe-t'il ensuite sur occurrence d'un événement `eY` ? Que se passe-t'il sur occurrence d'un événement `eZ`, l'état `EB` étant actif ?

5.3 Comportement de l'interface d'un téléphone

On reprend ici l'exercice sur la modélisation de l'interface d'un téléphone mobile sous forme d'un modèle états–transitions UML. On s'intéresse de nouveau à la fonctionnalité 'appel d'un correspondant'. L'objectif est maintenant d'utiliser des structures syntaxiques qui rendent le modèle plus compact, et donc plus lisible, et d'enrichir le modèle.

Q 5.1. Reprenez le modèle que vous avez déjà construit et regardez si vous ne pouvez pas en avoir une représentation plus simple en utilisant : des activités et transitions internes, des états hiérarchiques.

1. Appel d'un numéro du carnet d'adresses.
   Dans cette version plus élaborée du téléphone, on peut aussi appeler un correspondant, non pas en tapant son numéro, mais en le sélectionnant dans le carnet d'adresses stocké dans le téléphone (on n'arrête pas le progrès !). On ne détaille pas ici la manière dont on réalise cette opération.

   Q 5.2. Ajoutez cette nouvelle possibilité à votre modèle. Modifiez éventuellement le reste de votre modèle de façon à rendre ce genre d'ajout plus facile à l'avenir (on ne sait jamais avec le progrès...)

2. Enregistrement de l'appel dans le journal.
   À la fin d'un appel, le numéro qui vient d'être composé est enregistré dans le journal des appels sortants (avec la date et l'heure).

   Q 5.3. Ajoutez cette nouvelle fonctionnalité à votre modèle. (on vous avait prévenu au sujet du progrès...)

3. Gestion de l'éclairage lors de la composition d'un numéro.
   Toutes ces petites merveilles de la technologie ont un gros défaut, elles consomment beaucoup d'énergie et les batteries ont une faible autonomie. En particulier, l'affichage est

gourmand en énergie. On a donc décidé de diminuer l'intensité de l'éclairage quand on n'a pas sollicité l'interface du téléphone depuis un certain temps. L'appui sur une touche provoque le retour à l'intensité normale.

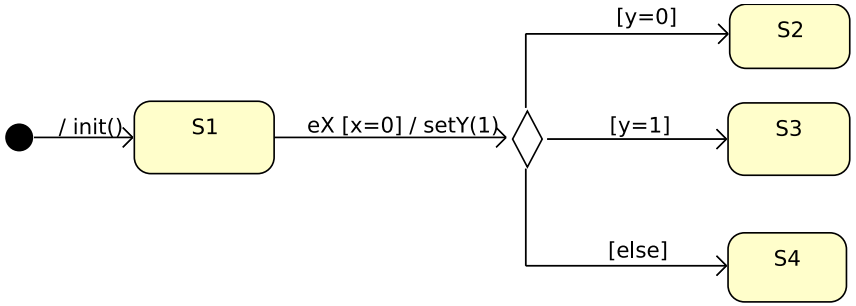
Q 5.4. Dans un premier temps, on n'implémente cet économiseur d'énergie uniquement lors de la composition du numéro. Modifiez votre modèle en conséquence.

Q 5.5. Maintenant, on généralise ce mode à l'ensemble des fonctionnalités du téléphone. Modifiez votre modèle en conséquence.

5.4 Alternatives

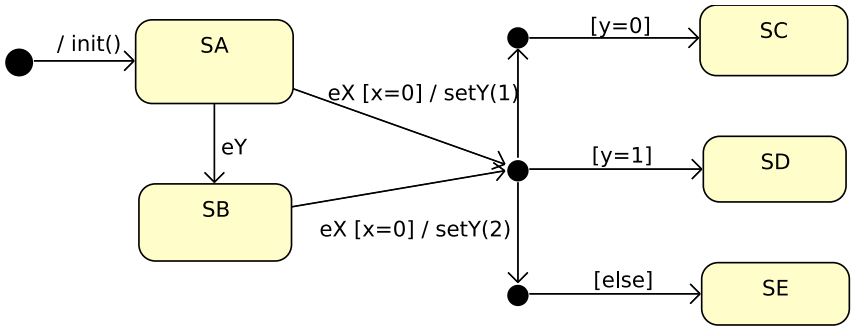
On considère le comportement suivant des objets d'une classe `CA` qui a un attribut `x: integer = 0` et `y: real` et deux opérations `init()` et `setY(in v: integer)`. La première affecte la valeur `0` aux attributs `x` et `y` ; la seconde affecte la valeur `v` à l'attribut `y`.

On associe la machine à état de la figure ci-dessous à la classe `CA`.



Q 5.1. Quel est l'état d'une instance de `CA` suite à l'occurrence d'un événement de type `eX` ?

On associe maintenant la machine à état de la figure ci-dessous à la classe `CA`.



Q 5.2. Quel est l'état d'une instance de `CA` suite à l'occurrence d'un événement de type `eX` ?
Même question pour un événement `eY` suivi de `eX`.

5.5 Hiérarchie

On considère ici la gestion des états des fenêtres d'une application, telle que l'on connait sur différents systèmes d'exploitation.

1. Les états fondamentaux d'une fenêtre sont : normale, réduite et agrandie (plein écran).
2. À la création de la fenêtre, celle-ci est ouverte, avec une taille normale.
3. Si la fenêtre n'est pas en plein écran, on peut la déplacer et changer ses dimensions.
4. Si elle est en plein écran et que l'on change la résolution de l'affichage, il faut la redimensionner.
5. Quand la fenêtre est ouverte, on peut l'agrandir, puis revenir à sa taille normale.
6. Étant ouverte, on peut la réduire ; si on l'ouvre à nouveau, la fenêtre reprend sa taille.

Q 5.1. Modélisez ce comportement sous forme d'un diagramme états–transitions UML. Vérifiez que, sous le système d'exploitation que vous utilisez, ce comportement est bien respecté. Corrigez éventuellement votre modèle.

6 Le modèle des activités : processus métier

Sommaire

6.1 Objectif

Ce travail porte sur le modèle des activités du langage UML :
- ▷ séquencement et alternatives ;
- ▷ flots de contrôle et de données ;
- ▷ partition des activités.

6.2 Modèlisation d'un processus métier

L'objectif est de modéliser un processus métier sous la forme d'un diagrame d'activités UML. L'étude porte sur le processus de développement d'un logiciel, plus particulièrement les phases suivantes : conception, réalisation (codage) et tests unitaires. Il ne s'agit que d'un cas d'école qui s'inspire librement de la réalité.

Première approche : la ligne droite.
Il s'agit ici de commencer par le cas le plus simple où la réalisation de chaque activité est un succès ; c'est le scénario sans échec, appelé parfois scénario nominal. Dans la suite, on envisagera les alternatives à ce scénario idéal.

1. La première étape est celle de l'édition des modèles ; elle se fait en phase de conception.
2. Quand la modélisation est terminée, on génère le code.
3. Le processus se poursuit par la production des exécutables et la mise au point du code (déroulement pas à pas du code au débugger, détection des éventuels problèmes d'allocation...). Ces activités s'inscrivent dans la phase de codage.
4. La dernière étape du processus, tel qu'il est envisagé ici, est celle de passation des tests unitaires, parfois réalisée par une équipe spécialisée.

Q.6.1 : Représentez cette première version du processus sous forme d'un diagramme d'activités UML.

Q.6.2 : Faites en sorte que l'organisation des activités soit bien visible en les partitionnant par phase.

Un processus plus itératif.
S'imaginer que l'on peut produire un modèle complet et sans erreur permettant une génération complète du code en une seule passe est un leurre. Dans la pratique, il est d'ailleurs préférable de mener la phase de conception en réalisant de courtes itérations sur les activités de modélisation et de codage [1].

1. L'activité de génération de code à partir du modèle se fait tant que la modélisation est considérée comme incomplète. Si le modèle est complet, alors on passe aux tests unitaires.
2. Le générateur de code effectue des vérifications sur la structure du modèle. Il peut, par exemple, détecter que le type d'un attribut n'a pas été spécifié... Si le modèle est incomplet, le générateur ne produit rien et il faut reprendre la modélisation ; sinon, on peut passer à la phase de codage.

Q.6.3 : Reprenez votre diagramme d'activités en introduisant ces alternatives. Utilisez pour cela les nœuds de décision et de fusion.

Procédures de test.
L'activité de test est délicate et coûteuse en temps. Pour la mener à bien, il faut rédiger au préalable des « spécifications de test » qui décrivent les programmes de test, les procédures de réalisation des tests, les données d'entrée et les résultats attendus.

1. La réalisation des tests suppose que l'on ait à sa disposition un exécutable et la spécification des tests.

1. À condition que les choix d'architecture soient au préalable validés, ce qui est hors du champ de cette modélisation.

2. Une fois les tests réalisés, si un écart est constaté entre les résultats attendus et ceux du test, alors on rédige un rapport d'anomalie. Ce rapport est une donnée d'entrée pour la conception du logiciel.

Q.6.4 : Reprenez votre diagramme d'activités en faisant figurer ces flots d'activités. Veillez à la cohérence des flots d'exécution.

7 Le modèle des activités : du modèle au programme

Sommaire

7.1 Objectif

Ce travail porte sur l'utilisation du modèle des activités du langage UML en phase de conception et de codage d'un logiciel. Il s'agit de voir comment on peut traduire un tel modèle en langage de programmation objet, ici en java.

Le cas d'étude est celui d'une version très simplifiée d'un simulateur de la vie d'animaux dans leur environnement : des lions, des gnous, des gazelles dans une savane.

7.2 Rétro-ingénierie : du code au modèle

Pour faire vivre chaque animal peuplant la savane, le simulateur, à chacun de ses cycles, appelle la méthode `live()` de la sous-classe d'`Animal`. Cette méthode correspond donc à l'exécution du comportement de l'animal à chaque pas de temps. Du manière générale, les animaux naissent, se déplacent, se nourissent, se reproduisent dans la savanne, et ils finissent par mourrir. La savanne est une étendue plane qui est discrétisée en cellules, `Cell2D`, pour les besoins de la simulation. Ces cellules peuvent être de différents types : couvertes d'herbe, `Grass` ou d'arbres, `Tree`. Le diagramme de la figure 1 présente les principales propriétés des classes sur lesquelles reposent les fonctionnalités du simulateur.

L'objectif ici est d'analyser le code de la méthode `live()` de la classe `Lion` et de le transcrire en un modèle d'activités. Vous trouverez le code de cette méthode dans l'archive disponible sur Moodle. Ce code est le résultat de la transformation du modèle de classes qui vous est fourni et qui a été produit en utilisant le logiciel de modélisation `bouml`.

Q.7.1 : Représentez la méthode `live` de la classe `Lion` sous forme d'un diagramme d'activités UML.

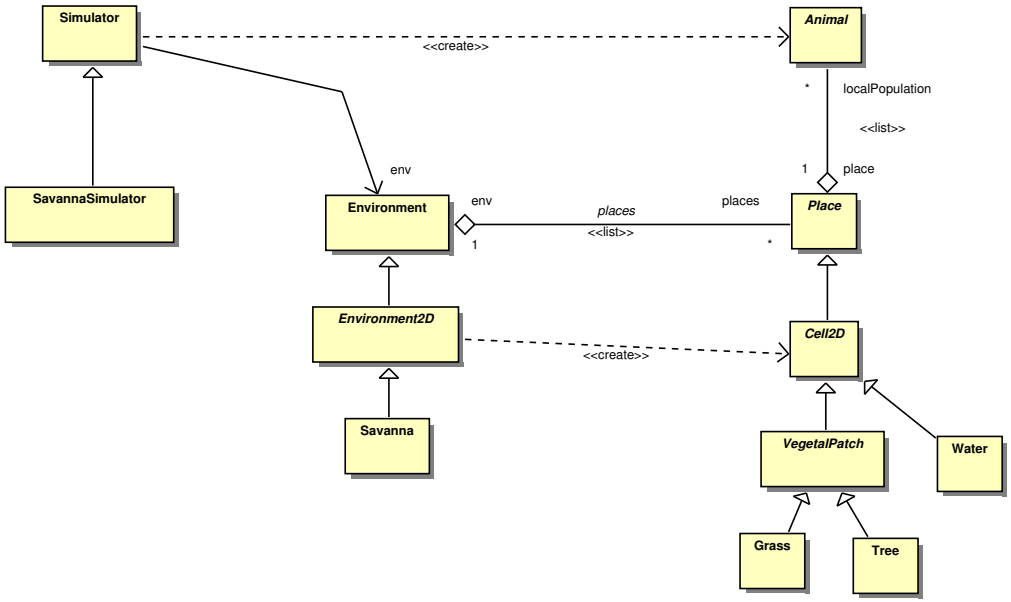


FIGURE 1 – Modèle de classes du simulateur de la vie des animaux dans la savane

7.3 Traduction d'un modèle en langage de programmation

On s'intéresse maintenant à un des comportements des lions qui est leur mode de chasse. Ce comportement est décrit sous forme d'un diagramme d'activité sur la figure 2.

La figure 3 donne l'ensemble des propriétés de la classe `Lion`. En cas de besoins, vous pouvez consulter le modèle sous bouml.

Q.7.1 : Écrivez le code de l'opération `hunt` de la classe `Lion` conformément au diagramme d'activités de la figure 2.

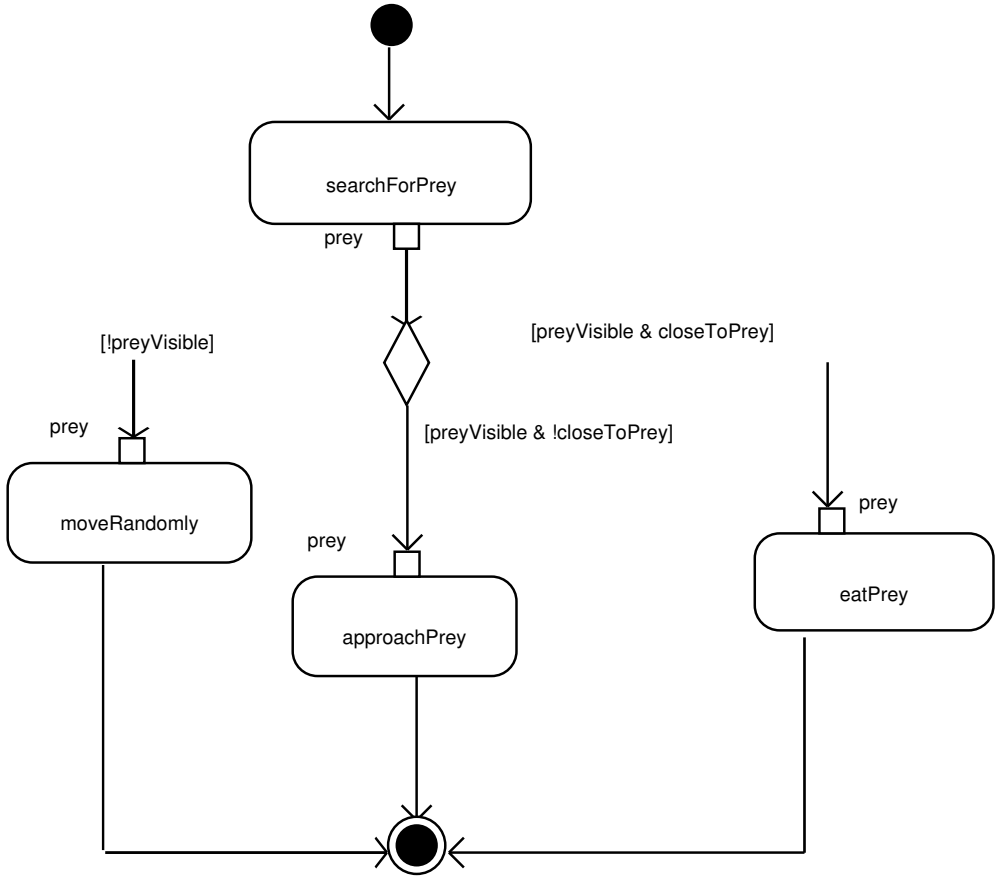


FIGURE 2 – Modèle d'activités de l'opération `hunt` de la classe `Lion`

FIGURE 3 – La classe Lion et ses dépendances

Solution Labo