Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2008/2009

Lesson 7 6 April 2009

Structured Knowledge Representation
Représentation Structurée de la Connaissance

Représentation Structurée de la Connaissance

Intelligence : (Petit Robert)

"La faculté de connaître et comprendre.

On a vu que Connaissance = Compétence.

Qu'est que c'est à comprendre?

Association entre choses perçues et choses connus.

Phénomène : tout ce qui est objet d'expérience possible. ,
Tout ce qui se manifeste par l'intermédiaire des sens.
(ref : Critique du Raison Pure, I. Kant, 1781)

Pour comprendre une scène ou une histoire, il faut trouver une correspondance entre les éléments perçus et les choses connus.

Les perceptions brutes (les phénomènes) sont compris par associé aux catégories mentales (les concepts). Cela exige une représentation. Une telle représentation et fourni par les Schémas

Schémas

Qu'est que c'est à comprendre?

Association entre choses perçues et choses connus.

Pour comprendre une scène ou une histoire, il faut trouver une correspondance
Entre les éléments perçus et les éléments qui représentent les connaissances. Les schemas fournissent une représentation déclarative pour la connaissance.

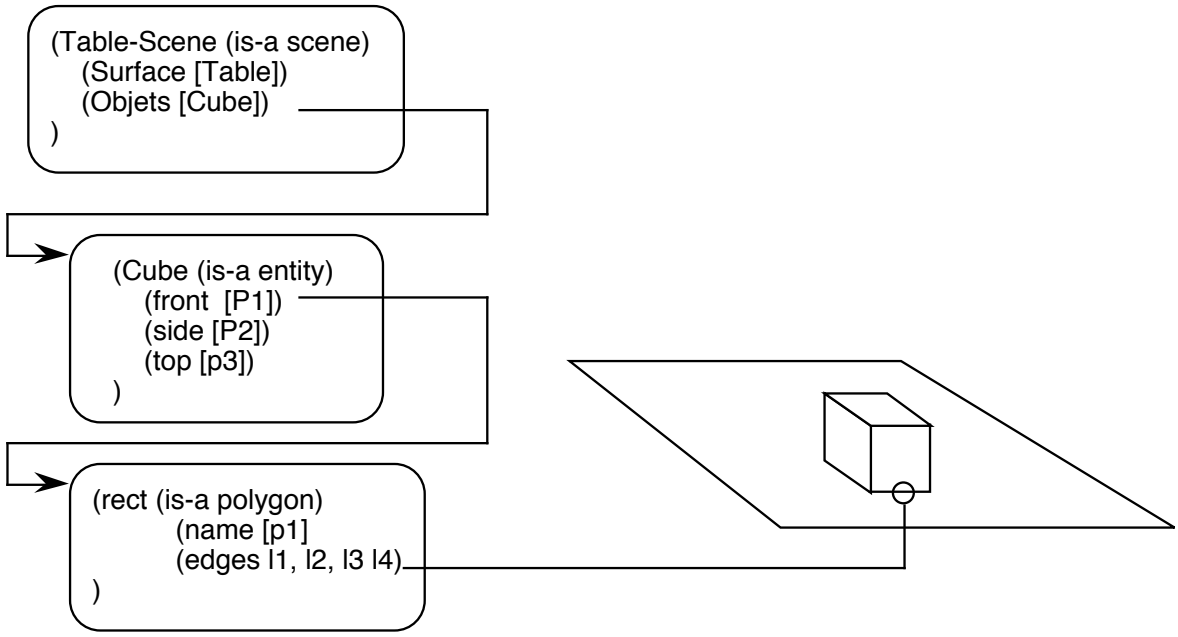
Elle permet de raisonner et comprendre les scènes, les histoires (orales ou écrites), les textes, les idées ou les plans.

L'expression de la connaissance déclarative par structure de schéma a plusieurs applications. Trois applications classique sont les Frames, les Scripts et les Réseaux Sémantiques.

Frames : Un assemblage des schémas pour décrire une scène ou un contexte.
Script : Un assemblage des schémas pour décrire une séquence des événements
Réseau Sémantique : Un assemblage des schémas pour décrire le sens sémantique d'un texte.

Frames

Le concept de "Frame" était proposé par Minsky pour guider l'interprétation d'une scène en vision par ordinateur. Un Frame guide l'interprétation d'une manière "top-down". Depuis les concepts à été généralisé, et maintenant on trouve le terme frame utilisé en synonyme avec "schéma".



Un frame est une description d'un contexte composé d'une collection de rôles et relations. Ceci donne une description "explicite" du contexte qui peut servir de guider l'interprétation s'une scène, d'une situation ou d'une histoire.

Les Frames spécifie un contexte pour guider l'interprétation dans un domaine. La structure d'une frame permet de faire les inférences par raisonnement par défaut.
Le domaine est composé d'entités.

Modèle de Situation :

Les Schémas sont une structure déclarative pour représenter des situations. Ils sont composés de entités et de relations. Pour leur définition, il faut définir quelques concepts fondamentaux.

Situation : Une configuration de relations entre les entités

Entité : Une association d'Observations.
L'association est typiquement par corrélation de position mais peut être par d'autres propriétés.

Relation : Un prédicat entre entités

Les situations peuvent être representé par les schémas

Les Relations

Les exemples de relations comprennent les positions relatives, les composants d'une structure, les relations de famille et les relations de temps.

La "Valence" ou "Arité" d'une relation est le nombre de choses associées.

Monadic ou unaire: Man(Bob)
dyadique (ou binaire): Brother(Charlie, Bob)
triadique (ou ternaire) : Action(Jim, Talks-To, Bob)

Les relations monadiques servent à représenter les propriétés.
Ils peuvent être représenté par la valeur dans un slot.

La relation dyadique associe deux objets. Ils peuvent être représenté par un pointer dans un slot.

Les relations triadiques exigent un nœud (un objet) avec les pointers pour les rôles.

Exemples : Dans le mondes de Blocs
Arity 0 : (HandEmpty)
Unary : (OnTable A)
Binary: (On A B)
Ternary: (Over A B C) ;; Block A is a bridge over B et C.

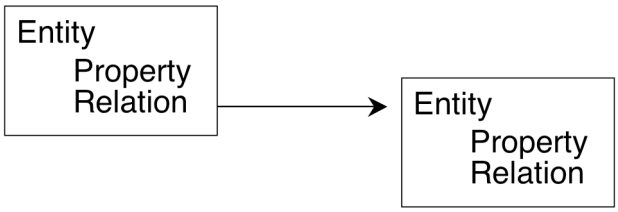
Exemple : Un symbole est une relation triadique entre une chose, un signe et un agent (un interprète)

```
(defclass symbol (is-a USER)
    (slot signe)
    (slot chose)
    (slot agent)
)
```

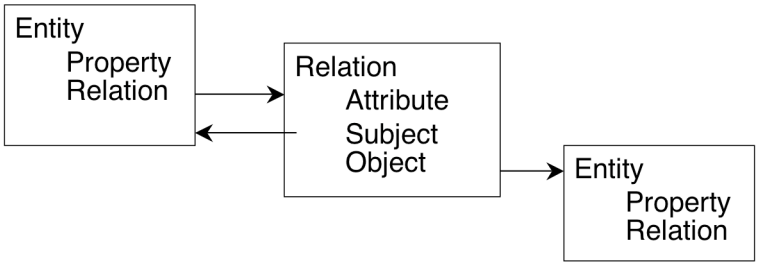
Comprendre un symboles voudrais dire trouver les entités pour chaque rôle.

Une relation peut être représenté d'un manière implicite (ex-valeur dans un slot) ou explicite (ex. par un objet de type "relation").
Les relations explicitent peuvent avoir un les attributes ainsi qu'un arity > 1.

Implicite :



Explicite :



Scripts

Un Script est une structure de schémas utilisés pour décrire une séquence d'événements. Un scripte est une forme de connaissance déclarative qui peut servir

1) Communiquer, comprendre et raisonner sur une histoire
2) Comprendre ou raisonner sur une situation ou une séquence d'événements
3) Raisonner sur les actions
4) Planifier les actions

Un scripte est composée de :

1) Une scène
2) Des "props" (Les objets manipulés dans le script).
3) Les acteurs (Les agents)
4) Les événements : Un changement de situation
5) Les Situations. Une configuration des entities et relations

Le scripte est une structure de scenes. Dans chaque scene, un ou plusieurs acteurs font les actions. Les acteurs agissent dans les lieus de la scène avec les props. Les situations peuvent être organisé en séquence, arbre ou réseau.

Le scripte aide à l'interprétation et à la communication par fournissant les éléments primitifs est les méthodes de leur reconnaissance.

En outre, le script permet de prédire les actions et événements possibles.

Exemples d'un script

1) Manger dans une bonne restaurant française

Scène : La (les) salle(s), l'entrée, les tables (comme lieu), la cuisine
Acteurs : Le maître d'Hôtel, Le serveur, la "bus-boy", les clients
Props : le menu la table (comme objet), les chaises, les couteaux, fourchettes et cuillers, la (les) verre(s)
situations: Entrer, S'asseoir, lire le menu, commander, manger, boire, demander l'addition, payer, partir.

La scène, les acteurs, le props, les actions, les situations sont tous représenté par les schéma.

2) Achat d'une boisson au distributeur automatique

Scène : devant la machine
Props: La machine, les pieces de monnaie, la boisson, le gobelet,
Acteur : acheteur
Actions : 1) Sortir tes pieces de monnaie
2) Payer
3) Sélectionner la boisson et les options (sucre, crème, etc.)
4) Recouper la boisson et le monnaie

Programmation des SCHEMA en CLIPS

Héritage des définitions des classes

Les classes et instances peuvent être organisés en hiérarchies qui permettent de définir les slots, les valeurs et les méthodes par défaut.

Héritage simple : Une superclass par classe
Héritage multiple : Plusieurs superclass par classe.

Héritage simple

Par exemple,

```
(defclass A  (is-a USER))
(defclass B  (is-a A))
(defclass C (is-a A))
(defclass D (is-a A))
```

Héritage multiple

<subclass> (is-a <superclass1> <superclass2> ... <superclass-n>)

S'il y a plus qu'une superclass, l'héritage est dit "multiple".

La liste "is-a" est le "**class precedence list**"
L'héritage donne priorité selon l'ordre dans cette liste.

Ordre : Spécifique -> Générale

Règle de l'héritage Multiple

1) Une classe à priorité sur ses superclass
2) Gauche à droite dans la liste "is-a"

L'héritage est appliqué à les superclass dans une sorte de recherche en profondeur d'abord.

Handlers : Communication par Message

La Communication par Message permet d'accéder d'une valeur dans un objet
La forme général d'un message est :

```
(SEND OBJET METHODE ARG*)
```

SEND : Mot Clé pour send (par fois c'est ¨) .
OBJET : Une pointer au objet, (dit l'objet "actif").
METHODE : la méthode à appeler (Get-Value ou autre).
ARG* : Les arguments pour la méthode.

Note : Il faut avoir un pointeur sur un objet pour l'accéder!

exemple :
(make-instance [Joe] of PERSON (name "joe")(age 32))
(send [Joe] get-name)

Les METHOD's pour les objets en CLIPS sont appelés des "message-handlers".
(nota: Il existe également les fonctions "generic" composés de "methods"
en CLIPS. Ceci nous ne concerne pas dans ce cours.)

Un certain nombre de message-handlers sont définis quand une classe
est déclarée d'être "concrete". Notamment : init, print, delete.

D'autre message-handlers sont définis pour les slots déclarés
avec des "create-accessor"

Dans le BNF on trouve :

<create-accessor-facet> ::= (create-accessor ?NONE | read | write | read-write)

Accessor Facet	Message Handler Crée.
read	get-<slot>
write	put-<slot>
read-write	get-<slot> et put-<slot>

Il est possible de déclarer d'autre message handlers, ainsi de changer la définition des message-handlers existant, avec "defmessage-handler".

Syntaxe :

```
(defmessage-handler  <class-name> <message-name>
                     [<handler-type>] [<comment>]
        (<parameter>* [<wildcard-parameter>])
         <action>*
)

<handler-type>      ::= around | before | primary | after
<parameter>         ::= <single-field-variable>
<wildcard-parameter>    ::= <multifield-variable>
```

Explication

<class-name> : Nom d'une classe
<message-name> : Nom du message handler
[handler-type] : Type de handler.
<parameters>: Variables clips
[wildcard-parameter] : variable du type liste
<action>: Les actions ou fonctions clips

exemples :

si (create-accessor read est déclaré :

```
(defmessage-handler <class> get-<slot-name> primary ()
    ?self:<slot-name>)
```

si (create-accessor write est déclaré :

```
(defmessage-handler <class> put-<slot-name> primary (?value)
    (bind ?self:<slot-name> ?value)
```

ou bien, si c'est un multi-slot :

```
(defmessage-handler <class> put-<slot-name> primary ($?value)
    (bind ?self:<slot-name> $?value)
```

?Self

Considère : (send [OBJ] fonction) L'objet [OBJ] est dit "actif".

Dans un message handler, la variable ?self fourni une pointer a l'objet actif.
On peut l'utiliser afin d'accéder aux slots.

Par exemple :

```
(defclass THING  (is-a USER) (role concrete)
   (slot NAME (create-accessor read-write) (default A))
)

(defmessage-handler THING ask-name ()
   (send ?self get-NAME)
)
```

?self donne un accès direct aux slots, avec la notation : ?self:<slot-name>.
ceci permet d'éviter le mécanisme de passage à message pour l'accès .

```
(defmessage-handler THING return-name ()
   ?self:NAME
)
```

NOTA : il est sans intérêt de faire :

(bind ?NAME (send ?self get-NAME))
ou même
(bind ?NAME ?self:NAME)

Utilisez
(?self:NAME)

Les types d'handlers

Handler Type	Class Role	Return
primary	Performs the majority of the work for the message	Yes
before	Does auxiliary work for a message before the primary handler executes	No
after	Does auxiliary work for a message after the primary handler executes	Yes
around	Sets up an environment for the execution of the rest of the handlers	No

Les types "before", "after" et "around" permettre de définir les "demons".

exemple :

```
(defmessage-handler THING get-NAME after ()
   (printout t "Quack" crlf)
)

get-name (primary) existe toujours.
```

exemple d'un demon Counter-demon

```
(defclass THING  (is-a USER) (role concrete)
   (slot NAME (create-accessor read-write) (default A))
   (slot COUNT (create-accessor read-write) (default 0))
)

(defmessage-handler THING get-NAME after ()
   (printout t "Quack" crlf)
)

(defmessage-handler THING get-NAME before ()
   (bind ?self:COUNT (+ ?self:COUNT 1))
   (printout t
            ?self:NAME " read "?self:COUNT " times." crlf)
)
```

Les handlers en CLIPS sont Polymorphique.
Un handlers pour une classe peuvent porter le même nom que des handlers d'autres classes.
L'handler exécuté est déterminé par la classe de l'objet actif.
Le même nom peut être donné à les handlers des classes différentes.

Activations des règles par objets

En Système Expert, les règles et les schèma sont complémentaire.

Depuis CLIPS 6, l'activation de regles par instances d'objet est possible.

Il faut declarer la classe de l'objet "(pattern-match reactive)".
Par exemple.

```
(defclass A (is-a USER)
   (role concrete)
   (pattern-match reactive)
   (slot foo  (create-accessor write)
              (pattern-match reactive)
   )
)
(make-instance a of A)
```

Les objet sont considère commes les faits d'une template "objet".
Dans le règle, on utilise le template "objet" avec le test de classe "is-a".

```
(defrule test-A-existe
   ?ins <- (object (is-a A))
=>
   (printout t "Objet " ?ins " est de la classe A" crlf)
)
```

On peut tester les valeurs de Slots dans les conditions

```
(defrule test-foo-eq-toto
   ?ins <- (object (is-a A) (foo ?f&~nil))
=>
   (printout t "Objet " ?ins " foo = " ?f crlf)
)
(run)
(send [a] put-foo toto)
(run)
```

Les regles peut servent, par exemple, a l'initialisation

```
(defclass PERSON (is-a USER) (role concrete)
             (pattern-match reactive)
             (slot famille  (create-accessor read-write))
             (slot prenom  (create-accessor read-write))
             (slot age (create-accessor read-write))
             (multislot address (create-accessor read-write))
)
```

```
(defrule demande-nom-de-famille
   ?ins <- (object (is-a PERSON) (famille nil))
=>
   (printout t "Quel est le nom de famille de "?ins "? ")
   (send ?ins put-famille (read))
)

(defrule demande-prenom
   ?ins <- (object (is-a PERSON) (prenom nil))
=>
   (printout t "Quel est le prenom de "?ins "? ")
   (send ?ins put-prenom (read))
)

(make-instance [Fred] of PERSON)
(make-instance [Bob] of PERSON (prenom Bob))
(run)
```

Des règles peut s'appliquer a les objets sans regarde pour leurs classes.
Un règle peut determiner le valeur de la classe.

```
(defclass STUDENT (is-a USER) (role concrete)
             (pattern-match reactive)
             (slot famille  (create-accessor read-write))
             (slot prenom  (create-accessor read-write))
             (slot age (create-accessor read-write))
             (slot option (create-accessor read-write))
             (slot promo (create-accessor read-write))
)


(make-instance [Bob] of PERSON (prenom Bob) (famille Barker)
(age 20))
(make-instance  [B] of STUDENT (prenom Bob) (famille Barker))


;;
;; Determiner la classe d'un objet
;;

(defrule deduire-class
   ?o <- (object (is-a ?c))
=>
  (printout t "l'objet " ?o " de classe "?c "." crlf)
)
;;
;; Completer un objet par un autre
;;
```

```
(defrule deduire-age
   ?o1 <- (object (famille ?f&~nil) (prenom ?p&~nil) (age
?a&~nil))
   ?o2 <- (object (is-a ?c) (famille ?f) (prenom ?p) (age nil))
=>
  (send ?o2 put-age ?a)
  (printout t "Affecter age " ?a " pour ")
  (printout t  ?c" " ?p " " ?f "." crlf)
```

Exemple : Les Relations de Famille

L'objet de cet exercice est de réaliser un système de classes permettant de répondre aux questions concernant les relations entre les membres d'une famille. Les réponses sont générées par des "handlers".

a) Définir la classe "personne" avec les slots "nom", "pere", "mere", "freres" et "soeurs" . Les slots frères et soeurs peuvent contenir une liste.

```
(defclass PERSON (is-a USER)
    (role abstract)
    (slot ID (create-accessor read-write))
    (slot father (create-accessor read-write) (default unknown))
    (slot mother (create-accessor read-write) (default unknown))
    (multislot brothers (create-accessor read-write))
    (multislot sisters (create-accessor read-write))
)
```

Définir la classe "homme" (sous-classe de personne) avec le slot "épouse" et le slot "sexe" avec une valeur fixe "masculin".

Définir la classe "femme" (sous-classe de personne) avec le slot "époux" et le slot "sexe" avec une valeur fixe "feminin".

```
;;
;;  Class pour personne, Homme et Femme
;;

(defclass MAN (is-a PERSON)
    (role concrete)(pattern-match reactive)
    (slot wife (create-accessor  read-write) (default unknown))
    (slot sex (storage shared)
            (default male) (create-accessor read))
)

(defclass WOMAN (is-a PERSON)
    (role concrete)(pattern-match reactive)
    (slot husband (create-accessor  read-write) (default unknown))
    (slot sex (storage shared)(access read-only)
```

```
            (default female) (create-accessor read))
)
```

Faire la règle pour demander les membres d'une famille

```
(defrule ask-wife
    ?M <- (object (is-a MAN) (ID ?n) (wife  unknown))
=>
    (printout t "Who is the wife of " ?n "? ")
    (bind ?ID (read))
    (send ?M put-wife ?ID)
    (if (neq ?ID nil) then
        (make-instance ?ID of WOMAN (ID ?ID) (husband ?n)))
)

(make-instance [Jean] of MAN (ID Jean))
(make-instance [Paul] of MAN (ID Paul))

(run 1)
(send [Paul] get-wife)

(defrule ask-father
    ?M <- (object (is-a MAN) (ID ?n) (father  unknown))
=>
    (printout t "Who is the father of " ?n "? ")
 (bind ?ID (read))
    (send ?M put-father ?ID)
 (make-instance ?ID of MAN (ID ?ID))
)
```

b) Définir des "message-handlers" pour la classe PERSON qui déterminent le "gandmere" et le "grandpere" paternels.

```
(defmessage-handler PERSON paternal-grandfather ()
    (send ?self:father get-father)
)
```

ou bien

```
(defmessage-handler PERSON paternal-grandfather ()
    (if (neq unknown ?self:father)
        then (send ?self:father get-father)
        else (printout t "the father of "?self:ID "is unknown"
CRLF)
    )
)

(defmessage-handler PERSON paternal-grandmother ()
    (send ?self:father get-mother)
)
```

c) Définir des "message-handlers" pour la classe PERSON qui déterminent les noms des "gandmere" et le "grandpere" paternels.

```
(defmessage-handler PERSON paternal-grandfather ()
    (bind ?g-father (send ?self:father get-father))
 (send ?g-father get-ID)
)

(defmessage-handler PERSON paternal-grandmother ()
    (bind ?g-father (send ?self:father get-mother))
 (send ?g-father get-ID)
)
```

d) Définir le message handler pour la classe PERSON qui détermine les oncles (freres du pere et freres de la mere). Aide : On peut composer une liste avec create$. Ex : (a b c) <- (create$ a b c)

```
(defmessage-handler PERSONNE uncles ()
    (create$ (send ?self:father get-brothers)
             (send ?self:mother get-brothers)
 )
)
```

e) Définir le message handler pour la classe PERSON qui détermine les noms des oncles (freres du pere et freres de la mere). Aide : On peut composer une liste avec create$. Ex : (a b c) <- (create$ a b c)

```
(defmessage-handler PERSON name-the-uncles ()
    (bind $?uncles
   (create$ (send ?self:father get-brothers)
            (send ?self:mother get-brothers)
   )
 )
 (progn$ (?uncle $uncles)
     (printout t "the names of " ?oncle " is ")
     (printout t (send ?Uncle get-ID) crlf)
  )
)

(defrule ask-brother
    ?M <- (object (is-a MAN) (ID ?ID) (brothers $?brothers))
    (test (eq (nth 1  $?brothers) unknown))
=>
    (printout t "Who is the brother of " ?ID "? ")
    (bind ?b (read))
    (if   (eq ?b nil) then (bind $?brothers (delete$ $?brothers 1
1))
        else (replace$ $?brothers 1 1 ?b))
    (send ?M put-brothers $?brothers))
)
```

f) Définir le message-handler pour la classe PERSON qui détermine la liste des noms de tous les grands-parents.

```
(defmessage-handler PERSON grand-parents ()
   (create$
      (send (send ?self:father get-father) get-ID)
      (send (send ?self:father get-mother) get-ID)
      (send (send ?self:mother get-father) get-ID)
      (send (send ?self:mother get-mother) get-ID)
   )
)
```

Représentation d'une Hiérarchie des Catégories

Une hiérarchie est une structure relationnelle entre entités. On peut organiser les hiérarchies entre les individus, les symboles, les concepts ou d'autres entités.

Par exemple, dans un réseau sémantique, les concepts sont organisés en hiérarchies. Le plus bas niveau est composé de mots. Une représentation hiérarchique permet une inférence des propriétés (par défaut) et une économie de représentation.

La relation d'abstraction ENTRE catégories est parfois appelée "AKO" (A Kind of). Nous allons faire une inférence entre membres de d'une hiérarchie par une sorte d'"héritage dynamique".

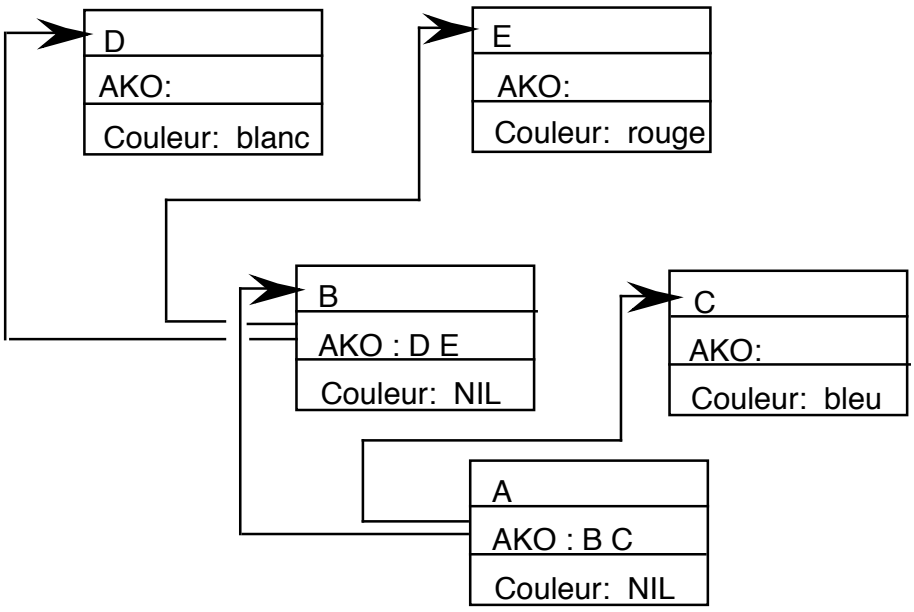
On peut trouver l'héritage simple (Une AKO par concept) ou l'héritage multiple (Plusieurs AKO par classe).

L'héritage dynamique est une forme de parcours de graphe.
Il peut être multiple. Il peut être "strict" ou "defeasible".
Héritage multiple "defeasible" permet les exceptions.

Exemple:

```
(defclass CHOSE (is-a USER) (role concrete)
   (multislot AKO (create-accessor read-write))
   (slot couleur (create-accessor read-write))
)

(make-instance [A] of CHOSE (AKO [B] [C]))
(make-instance [B] of CHOSE (AKO [D] [E]))
(make-instance [C] of CHOSE (couleur bleu))
(make-instance [D] of CHOSE (couleur blanc))
(make-instance [E] of CHOSE (couleur rouge))
```



La Relation AKO est réalisée par une “message-handler”.

```
(defmessage-handler CHOSE inherit-couleur  ()
   (if (neq ?self:couleur nil) then (return ?self:couleur)
   else
      (bind ?couleur)
      (progn$ (?super ?self:AKO)
         (bind ?couleur (send ?super inherit-couleur))
         (if (neq nil ?couleur) then (break))
      )
      (return ?couleur)
   )
)

Une message-handler pour une héritage multiple est

;;
;; héritage multiple. Les valeurs sont aussi affecté aux slots.
;;

(defmessage-handler CHOSE mv-inherit-couleur  ()
   (if (neq ?self:couleur nil) then  (return ?self:couleur)
   else
      (bind $?couleur (create$))
      (progn$ (?super ?self:AKO)
          (bind ?c (send ?super inherit-couleur))
          (if (neq ?c nil) then
            (bind $?couleur (insert$ $?couleur 1 ?c))
          )
      )
      (return $?couleur)
   )
)
```