CLIPS : 2 - Objets et classes, Clips Object Oriented Language

(C Language Integrated Production System)

Bernard ESPINASSE
Professeur à l'Université d'Aix-Marseille

Plan

- Introduction à COOL
- Classes systèmes prédéfinies
- Le construct DEFCLASS
- L'héritage multiple
- Les règles de Clips
- Le contrôle dans Clips

Introduction

- CLIPS provides 3 paradigms:
  - **rules**,
  - **objects** : CLIPS ObjectOriented Language (COOL) which is integrated with the rule and procedural based paradigms of CLIPS
  - **procedures** : generic functions, deffunctions, and user-defined external functions
- Depending on the application, you can use rules, objects, procedures, or a combination.
- **in object paradigm** the 5 generally accepted features of a true OOP language are :
  - abstraction,
  - inheritance,
  - encapsulation,
  - polymorphism and
  - dynamic binding
- **ces 5 caractéristiques sont assurées dans Clips**

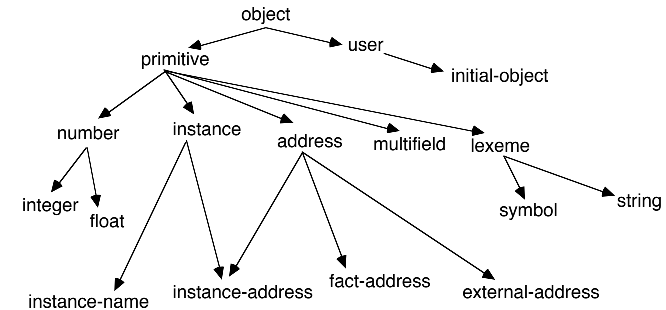
Objets et Classes dans Clips

- un **objet** est composé :
  - de **propriétés** (**slots**)
  - d'un **comportement** (**message-handler**)
- un **objet** est défini pour être :
  - une chaîne de caractères (string)
  - un nombre (float ou integer)
  - une valeur multi-champs
  - une adresse externe
  - une instance d'une classe définie par l'utilisateur
- une **classe** est une structure (template) pour des propriétés et des comportements communs d'objets
- un **objet** est une **instance** d'une **classe**

Objets et Classes dans Clips

- **2 types d'objets en Clips :**
  - objets de **type primitif** (primitive types)
    - référencés en donnant leur valeur
    - sans nom et sans slot
    - sont utilisés dans des fonctions génériques
  - objets qui sont des **instances de classes utilisateur** (instances of user-defined classes)
    - **référencés** par son nom ou son adresse
    - **créés** ou **détruits** explicitement par **messages** et **fonctions spéciales**
    - leurs **propriétés** s'expriment au travers de slots **définis** dans la **classe** de l'objet
    - les slots = **champs nommés** simples ou mutiples
    - leurs **comportement**s sont spécifiés en terme de code procédural applé "**message handlers**" attaché à la classe de l'objet
    - lorsqu'ils appartiennent à une même classe ils ont le **même ensemble de slots** pouvant avoir des valeurs différentes

Les 17 classes système de Clips



- toutes les classes sont des classes abstraites (seulement utilisées pour l'héritage) sauf **initial-object,** seule classe concrète et réactive aux prémisses des règles
- seule la classe **user** possède des **slots** et des **message-handlers**
- la classe **object** est une **superclasse**
- toutes les classes définies par l'utilisateur héritent de la classe **user**
- les classes **instance**, **instance-address** et **instance-name** sont essentiellement destinées à être utilisées par les **méthodes** (n'ont pas de sous-classes)

Objets et Classes dans Clips

Exemples d'objets et de leur classes :

Objets (Printed Representation)	Classe
Rolls-Royce	SYMBOL
"Rolls-Royce"	STRING
8.0	FLOAT
8	INTEGER
(8.0 Rolls-Royce 8 [Rolls-Royce])	MULTIFIELD
<Pointer- OOCF61 AB>	EXTERNAL-ADDRESS
[Rolls-Royce]	CAR (a user-defined class)

Le construct **defclass**

- permet à l'utilisateur de définir les **slots** (propriétés) et **message-handlers** (comportements) des classes
- 4 éléments :
  1) a name,
  2) a list of superclasses from which the new class inherits slots and message-handlers,
  3) a specifier saying whether or not the creation of direct instances of the new class is allowed and
  4) a list of slots specific to the new class.
- All user-defined classes must inherit from at least one class, and to this end COOL provides predefined system classes for use as a base in the derivation of new classes.
- Any slots explicitly given in the defclass override those gotten from inheritance. COOL applies rules to the list of superclasses to generate a class precedence list for the new class. Facets further describe slots. Some examples of facets include: default value, cardinality, and types of access allowed.
- the object's slot values cannot be altered or examined without sending a message to the object.

Le construct defclass

• **syntaxe :**

```
(defclass <name> [<comment>]
  (is-a <superclass-name>+)
  [<role>]
  [<pattern-match-role>]
  <slot>*
  <handler-documentation>*)

<role>  ::= (role concrete | abstract)
<pattern-match-role> ::= (pattern-match reactive | non-reactive)
<slot>  ::= (slot <name> <facet>*) |(single-slot <name> <facet>*) |
            (multislot <name> <facet>*)

<facet> ::=  <default-facet> | <storage-facet> | <access-facet> | <propagation-facet> |
             <source-facet> | <pattern-match-facet> | <visibility-facet> |
             <create-accessor-facet> | <override-message-facet> | <constraint-attributes>
<default-facet> ::=
           (default ?DERIVE | ?NONE | <expression>*) |
           (default-dynamic <expression>*)

<storage-facet> ::= (storage local | shared)
<access-facet>  ::= (access read-write | read-only | initialize-only)
<propagation-facet> ::= (propagation inherit | no-inherit)
<source-facet> ::= (source exclusive | composite)
<pattern-match-facet> ::= (pattern-match reactive | non-reactive)
<visibility-facet> ::= (visibility private | public)
<create-accessor-facet> ::= (create-accessor ?NONE | read | write | read-write)
<override-message-facet> ::= (override-message ?DEFAULT | <message-name>)
<handler-documentation> ::= (message-handler <name> [<handler-type>])
<handler-type> ::= primary | around | before | after
```


Exemple

- **An object in CLIPS is an instance of a class.**
- **An instance is an object** that has **values** for the **slots** such as : "John Smith, 28, 1000 Main St., Clear Lake City, TX"
- Lower-level classes automatically inherit their slots from higher-level classes, unless the slots are explicitly blocked.
- New slots are defined in addition to the inherited slots to set all the attributes that describe the class.
- **An object's behavior** is defined by its messagehandlers, or handlers for short. A message-handler for an object responds to messages and performs the required actions. For example, sending the message :

```
(send [John_Smith] print)
```

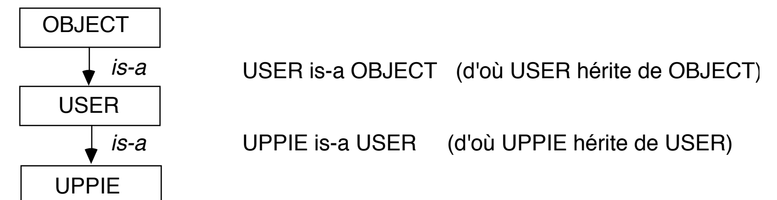
would cause the appropriate message-handler to print the values of the slots of the instance John_Smith. Instances are generally specified within brackets, [ ]. A message begins with the send function, followed by the instance name, message name, and any required arguments (in the case of the print message, there are no arguments)

Objets et héritage

- la différence entre un objet instance de classe et un template (fait non ordonné) est la notion **d'héritage**
- l'héritage permet que les **propriétés** et les **comportements** d'une classe peuvent être **décrits en termes d'autres classes**
- **COOL** (Clips Oriented Object Languages) supporte **l'héritage multiple** : une classe peut hériter directement de slots et de message-handlers de **plusieurs** classes

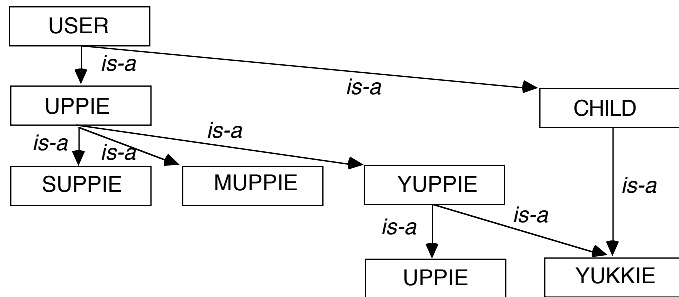
Exemple de hiérarchie de classes

- définissons une classe nommée UPPIE (urban professional) de la façon suivante :



- Remarque : les classes sont toujours nommées en majuscules les fonctions et commandes toujours évoquées en minuscules
- il y a héritage selon les liens "is-a"

Exemple de hiérarchie de classes



- légende :
  - UPPIE : Urban Professional (UP)
  - YUPPIE : Young UP
  - MUPPIE : Middle age UP
  - SUPPIE : Senior UP
  - PUPPIE : Prégnant UP
  - YUKKIE : kid of UP

Exemple de hiérarchie de classes

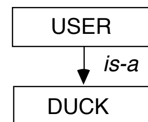
- déclaration des classes (defclass) :

```
CLIPS > (defclass UPPIE (is-a USER))
CLIPS > (defclass CHILD (is-a USER))
CLIPS > (defclass SUPPIE (is-a UPPIE))
CLIPS > (defclass MUPPIE (is-a UPPIE))
CLIPS > (defclass YUPPIE (is-a UPPIE))
CLIPS > (defclass PUPPIE (is-a YUPPIE))
CLIPS > (defclass YUKKIE (is-a YUPPIE CHILD))
```

- fonctions d'édition de la hiérarchie de classes :
  - `list-defclasses` : liste l'ensemble de toutes les classes
  - `browse-classes` : donne la hierarchie de classes globale ou à partir d'une classe donnée avec indentation
  - `superclassp` : permet de vérifier si une classe est la super classe d'une classe
  - `subclassp` : permet de vérifier si une classe est la sous classe d'une classe
  - `subclassp` : Pretty-print the defclass internal structure
  - `pdefclass` : Pretty-print the defclass internal structure
  - `undefclass` : Eliminate class
  - `describe-class` : Additional information about classes
  - `class-abstractp` : Predicate function returns TRUE if the class is abstract
- exemples :

```
•CLIPS > (superclassp UPPIE YUPPIE)
 TRUE
•CLIPS > (subclassp UPPIE YUPPIE)
 FALSE
```


Création et détails d'une classe



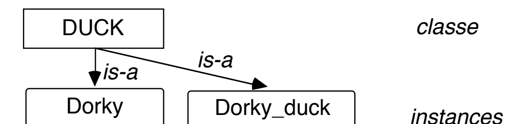

```
CLIPS> (defclass DUCK (is-a USER) (role concrete))
CLIPS> (describe-class DUCK)
================================================================
****************************************************************
Concrete: direct instances of this class can be created.
Non-reactive: direct instances of this class cannot match defrule patterns.

Direct Superclasses: USER
Inheritance Precedence: DUCK USER OBJECT
Direct Subclasses:
--------------------------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
CLIPS>
****************************************************************
================================================================
CLIPS>
```


Création d'instances d'un objet : fonction make-instance

- Création d'instances d'un objet par la fonction **make-instance :**

```
(make-instance [<instance-name>] of <class> <slot-override>)
```




```
CLIPS> (make-instance [Dorky] of DUCK)
[Dorky]
CLIPS> (make-instance [Elsie] of COW)
[PRNTUTIL1] Unable to find class COW.
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

- **règles :**
  - Only one instance of the same name may be used in a module.
  - A class cannot be redefined if instances of the class exist
  - Après un RESET, les instances créés avec make-instance sont détruites

Création d'instances d'un objet : fonction make-instance

- Just as (deffacts) defines facts, there is also a definstances to define instances when a (reset) is issued

```
<definstances-construct> ::= (definstances <definstances-name> [<comment>] <instance-template>*)
```

- exemple :

```
CLIPS> (definstances DORKY_OBJECTS "The Dorky Cousins"
    (Dorky of DUCK)
    (Dorky_Duck of DUCK))
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
For a total of 1 instance.
CLIPS> (reset)
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 3 instances.
CLIPS>
```


Destruction d'instances d'objet : fonction unmake-instance

- If you want to permanently delete an instance, the function unmake-instance will delete one or all instances, depending on its argument. To delete all instances, use the "*".

```
(unmake-instance <instance-expression> | *)
```

- exemple :

```
CLIPS> (unmake-instance *)      ; Delete all instances
TRUE
CLIPS> (instances)                      ; Check that all are gone
CLIPS> ; Restore the definstance
(definstances DORKY_OBJECTS "The Dorky Cousins"
    (Dorky of DUCK)
    (Dorky_Duck of DUCK))
CLIPS> (reset)                          ; Create new instances again
CLIPS> (instances)                      ; Check new instances created
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 3 instances.
CLIPS> (unmake-instance [Dorky]) ; Delete a specific instance
TRUE
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```


Destruction d'instances par envoi de message : fonction send

- Another way to delete a specific instance is to **send** a **delete message**
- The general syntax of the (send) function is as follows :

```
(send [<instance-name>] <message>)
```

- Only one instance name can be specified in a command and it must be surrounded by brackets if it is a user-defined name.
- exemple :

```
CLIPS> (reset)                          ; Create new instances again
CLIPS> (instances)                      ; Check new instances created
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 3 instances.
CLIPS> (send [Dorky_Duck] delete)
TRUE
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
For a total of 2 instances.
CLIPS>
```


Destruction d'instances par envoi de message : fonction send

- The "*" in a (send) will not work to delete all instances. The "*" only works with the (unmake) function. Another alternative is to define your own handler for delete that will accept the "*" and thus allow you to (send [instance-name] my_delete *) messages.

A (send) message is acted upon only by a target object which has an appropriate handler. CLIPS automatically provides handlers for print, init, delete and so on for each user-defined class.

It's important to realize that the message (send [Dorky_Duck] delete) works only because this instance is a user-defined class.

If you define classes which do not inherit from USER such as a subclass of INTEGER, you must also create appropriate handlers to carry out all desired tasks such as printing, creating, and deleting instances. It's much easier to define subclasses of USER and take advantage of system-supplied handlers.

Envois de messages : autre exemple

- The (send) function is the only proper way for objects to communicate
- According to the principle of object encapsulation, one object should only be allowed to access another object's data by sending a message.
- exemple : One useful application of (send) is to print information about an object :

```
CLIPS> (clear)
CLIPS> (defclass DUCK (is-a USER) (role concrete)
   (slot sound (create-accessor read-write))
   (slot age (create-accessor read-write)))
CLIPS> (definstances DORK_OBJECTS
   (Dorky_Duck of DUCK))
CLIPS> (reset)
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS> (send [Dorky_Duck] put-sound quack)
quack
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(age nil)
CLIPS>
```

- note that the slots are printed in the order defined in the class.
- However, if the instance inherits slots from more than one class, the slots from the more general classes will be printed first.

Modification d'instance : fonction modify-instance

- The value of a slot can be set by a slot-override in a make-instance. As an example,

```
CLIPS> (make-instance Dixie_Duck of DUCK (sound quack) (age 2))
[Dixie_Duck]
CLIPS> (send [Dixie_Duck] print)
[Dixie_Duck] of DUCK
(sound quack)
(age 2)
CLIPS>
```

- The complementary message to "put-" is get- which gets the data from a slot, as shown in the following example.
  - If a put- is successful it returns the new value, while if get- is successful it returns the appropriate data.
  - If the put- or get- does not succeed, an error message will be returned. The following examples show how this works.

```
CLIPS> (send [Dorky_Duck] put-color white)      ; No slot color
[MSGFUN1] No applicable primary message-handlers found for put-color.
FALSE
CLIPS> (send [Dorky_Duck] get-age)
nil
CLIPS> (send [Dorky_Duck] put-age 1); Value put in age
1
CLIPS> (send [Dorky_Duck] get-age); Check value is correct
1
CLIPS>
```


A very important point about slots is that you cannot modify the slots of a class by adding slots, deleting slots, or changing the characteristics of slots.

The only way to change a class is to :

- (1) delete all instances of the class, and
- (2) use a (defclass) with the same class name and the desired slots.

This situation is analogous to modifying a rule by loading in a new rule with the same name.

Fonctions de manipulation de slots

- If you use these predicate functions to test for appropriate values to functions, your program will be more robust against crashes.
  - `classslotexistp` : Returns TRUE if the class slot exists
  - `slot-existp` : Returns TRUE if the instance slot exists
  - `slot-boundp` : Returns TRUE if the specified slot has a value
  - `instance-address` : Returns the machine address at which the specified instance is stored.
  - `instance-name` : Returns the name given an address and vice versa
  - `instancep` : Returns TRUE if its argument is an instance
  - `instanceaddressp` : Returns TRUE if its argument is an instance address
  - `instance-namep` : Returns TRUE if its argument is an instance name
  - `instance-existp` : Returns TRUE if instance exists
  - `list-definstances` : Lists all the definstances
  - `ppdefinstances` : Pretty-prints the definstance
  - `watch instances` : Allows you to watch instances being created and deleted.
  - `unwatch instances` : Turns off watching instances.
  - `save-instances` : Save instances to a file
  - `load-instances` : Load instances to a file
  - `undefinstances` : Deletes the named definstance
- In general, if a function does not return TRUE, it returns FALSE.

Les facettes

- In this chapter you'll learn more about slots and how to specify their characteristics by using facets.
- The use of facets is good software engineering because there is a greater chance of CLIPS flagging an illegal value rather than risking a runtime error or crash.
- There are many types of facets that may be used to specify slots, as summarized in the following table.

Facet Name Description :

default and default-dynamic	: Set initial values for slots
cardinality	: Number of multifield values
access type	: Read-write, read-only, initialize-only access
storage	: Local slot in instance or shared slot in class
propagation	: Inherit, or no inherit slots
source	: composite or exclusive inheritance
documentation	: Documentation of slots
override-message	: Indicate message to send for slot override
create-accessor	: Create put- and get- handlers
visibility	: Public, or private to defining class only
reactive	: Changes to a slot trigger pattern- matching

Objets initiaux : le construct **definstance**

- similaire au construct **deffact**
- permet de spécifier un ensemble de connaissances initiales ou a priori en tant que collection d'instances de classes utilisateur
- après un **reset**, chaque instance spécifiée par un construct definstance est ajouté à la **liste des instances**
- and the equivalent of a make-instance function call is made for every instance specified in definstances constructs.
- **Syntaxe :**

```
<definstances-construct>
                   ::= (definstances <definstances-name> [<comment>]
                           <instance-template>*)
<instance-template>    ::= (<instance-definition>)
<instance-definition>  ::= <instance-name-expression> of
                                 <class-name-expression>
                                 <slot-override>*
<slot-override>        ::= (<slot-name-expression> <expression>*)
```

- **Exemple :**

....

Le construct **defmessage-handler**

- les objets sont manipulés par des **envoies de message** via la fonction **send**
- le résultat d'un message est une **valeur de retour** ou un **effet de bord**
- **defmessage-handler** permet de spécifier le comportement d'une classe d'objets en réponse à un message particulier
- l'implémentation du message est réalisé par une portion de code procédural appelé "message-handler" ou "handler"
- dans la hiérarchie de classe, chaque classe traite par des handlers une partie du message
- 7 éléments :
  1) a class name to which to attach the handler (the class must have been previously defined),
  2) a message name to which the handler will respond,
  3) an optional type (the default is primary),
  4) an optional comment,
  5) a list of parameters that will be passed to the handler during execution,
  6) an optional wildcard parameter and
  7) a series of expressions which are executed in order when the handler is called. The return-value of a message-handler is the evaluation of the last expression in the body.

Le construct **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>
```

Le construct defmodule

- CLIPS provides support for the modular development and execution of knowledge bases with the defmodule construct.
- CLIPS modules allow a set of constructs to be grouped together such that explicit control can be maintained over restricting the access of the constructs by other modules.
- This type of control is similar to global and local scoping used in languages such as C or Ada. By restricting access to deftemplate and defclass constructs, modules can function as blackboards, permitting only certain facts and instances to be seen by other modules. Modules are also used by rules to provide execution control.
- syntaxe :

```
<defmodule-construct> ::= (defmodule <module-name> [<comment>]
                             <port-spec>*)
<port-specification>  ::= (export <port-item>) |
                          (import <module-name> <port-item>)
<port-item>           ::= ?ALL |
                          ?NONE |
                          <port-construct> ?ALL |
                          <port-construct> ?NONE |
                          <port-construct> <construct-name>+
<port-construct>      ::= deftemplate | defclass |
                          defglobal | deffunction | defgeneric
```


Les contraintes d'attributs

- can be associated with deftemplate and defclass slots so that type checking can be performed on slot values when template facts and instances are created. The constraint information is also analyzed for the patterns on the LHS of a rule to determine if the specified constraints prevent the rule from ever firing.
- syntaxe :

```
<constraint-attribute> ::= <type-attribute>†|
                           <allowed-constant-attribute> |
                           <range-attribute> |
                           <cardinality-attribute>
                           <default-attribute>
<type-attribute>       ::= (type <type-specification>)
<type-specification>   ::= <allowed-type>+ | ?VARIABLE
<allowed-type>         ::= SYMBOL | STRING | LEXEME | INTEGER | FLOAT | NUMBER |
                           INSTANCE-NAME | INSTANCE-ADDRESS |
                           INSTANCE | EXTERNAL-ADDRESS | FACT-ADDRESS
<allowed-constant-attribute>
                       ::= (allowed-symbols†<symbol-list>) |
                           (allowed-strings <string-list>) |
                           (allowed-lexemes <lexeme-list> |
                           (allowed-integers†<integer-list>) |
                           (allowed-floats†<float-list>) |
                           (allowed-numbers†<number-list>) |
                            (allowed-instance-names <instance-list>)|
                           (allowed-values†<value-list>) |
<symbol-list>          ::= <symbol>+ | ?VARIABLE
<string-list>          ::= <string>+ | ?VARIABLE
```



```
<lexeme-list>           ::= <lexeme>+ | ?VARIABLE
<integer-list>          ::= <integer>+ | ?VARIABLE
<float-list>            ::= <float>+ | ?VARIABLE
<number-list>           ::= <number>+ | ?VARIABLE
<instance-name-list>    ::= <instance-name>+ | ?VARIABLE
<value-list>            ::= <constant>+ | ?VARIABLE
<range-attribute>       ::= (range <range-specification>
                                   <range-specification>)
<range-specification>   ::= <number> | ?VARIABLE
<cardinality-attribute> ::= (cardinality <cardinality-specification>
                             <cardinality-specification>)
<cardinality-specification> ::= <integer> | ?VARIABLE
```


Les contraintes d'attributs

- can be associated with deftemplate and defclass slots so that type checking can be performed on slot values when template facts and instances are created. The constraint information is also analyzed for the patterns on the LHS of a rule to determine if the specified constraints prevent the rule from ever firing.

Du bon usage des classes en Clips

The proper use of classes is summarized in the following three rules (Rules of Class Etiquette) :

1. The class hierarchy should be in specialized logical increments using is-a links.
2. A class is unnecessary if it has only one instance.
3. A class should not be named for an instance and vice versa.

The first rule discourages the creation of a single class for your application. If a single class is adequate, then you probably don't need OOP. By creating classes in increments, you can more easily verify, validate, and maintain your code. In addition, incremental class hierarchies can be easily put in class libraries to greatly facilitate the creation of new code. This concept of class libraries is analogous to subroutine libraries for actions. Only is-a links can be used since this is the only relationship that Version 6.0 of CLIPS supports.

The second rule encourages the idea that classes are intended as a template to produce multiple objects of the same kind. Of course you can start out with zero or one instance. However, if you'll never need more than one instance in a class, you should consider modifying its superclass to accomodate the instance rather than defining a new subclass. If all your classes only have one instance, it is probable that your application is simply not well-suited to OOP and that coding in a procedural language may be best.

The third rule means that classes should not be named after instances and vice versa, to eliminate confusion.

Les facettes

- permettent de spécifier des caracteristiques des slots
- principales facettes :

default and default-dynamic	: valeurs de slots par défaut
cardinality	: nombre de valeurs multiples
acess type	: accés en lecture-écrit., lecture seule, initialisation seule
storage	: slot local en instance ou slot partagé en classe
propagation	: slots d'héritage ou non
source	: héritage composite ou exclusif
documentation	: documentation du slot
override-message	: indique le message à envoyer pour un slot override
create-accessor	crée des put- et get- handers
visibility	publique ou privée dans la définition de classe
reactive	change à un slot trigger pattern-maching

Exemple : la facette "default"

- A Slot Named DefaultThe default facet sets the default value of a slot when an instance is created or initialized
- example :

```
CLIPS> (clear)
CLIPS> (defclass DUCK (is-a USER)(role concrete)
   (slot sound (create-accessor read-write)(default quack))
   (slot ID (create-accessor read-write))
   (slot sex (create-accessor read-write)(default male)))
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID nil)
(sex male)
CLIPS>
```

- This default facet is a static default because the value of its facet expression is determined when the class is defined and never changed unless the class is redefined.

Exemple : la facette "default-dynamic"

```
CLIPS> (clear)
CLIPS> (defclass DUCK (is-a USER)(role concrete)
   (slot sound (create-accessor read-write)(default quack))
   (slot ID (create-accessor read-write)
               (default-dynamic (gensym*)))
   (slot sex (create-accessor read-write)(default male)))
CLIPS> (make-instance [Dorky_Duck] of DUCK) ; Dorky_Duck #1
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen2)
(sex male)
CLIPS> (make-instance [Dorky_Duck] of DUCK) ; Dorky_Duck #2
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen3)              ; Note ID is different from Dorky_Duck #1
(sex male)
CLIPS>
```

- In this example which uses dynamic default, the ID of the second instance, gen3, is different from the first instance, gen2.
- In contrast, for the previous example of static default, the ID values were the same, gen1, since the (gensym*) was only evaluated once when the class was defined rather then for every new instance in the case of dynamic default.

La facette "multislot"

- permet de stocker des valeurs à champs multiples

```
CLIPS> (clear)
CLIPS> (defclass DUCK (is-a USER)(role concrete)
    (multislot sound (create-accessor read-write)(default quack quack))
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack quack)
CLIPS>
```

- une valeur à champs multiples peuvent être accedées par **get-** et **put- :**

```
CLIPS> (send [Dorky_Duck] put-sound quark1 quark2 quark3)
quark1 quark2 quark3
CLIPS> (send [Dorky_Duck] get-sound)
quark1 quark2 quark3
CLIPS>
```

- la fonction **nth$** permet d'obtenir le nth champs d'une valeur à champs multiples :

```
CLIPS> (nth$ 2 [Dorky_Duck] get-sound)
quark2
CLIPS>
```

- autres fonctions utiles :
  - **slot-replace$** : remplace le champs spécifié
  - **slot-insert$** : insère le champs spécifié
  - **slot-delete$** : éfface le champs spécifié

Manipuler les Handlers

- Handlers are essential in OOP because they support object encapsulation
- The only proper way that objects can respond to messages is by having an appropriate handler to receive the message and take appropriate action.
- In this chapter you'll learn the how messages are interpreted by objects. You'll see how to modify existing message-handlers, and how to write your own.

```
CLIPS> (defmessage-handler NUMBER + (?arg)    ; Argument of handler
            (+ ?self ?arg))                   ; Function addition of handler
CLIPS> (send 1 + 2)
3
```

- The variable **?self** is a special variable in which CLIPS stores the **active instance•** The ?self (reserved word) cannot be explicitly included in a handler argument, nor can it be bound to a different value.
- The active instance is the instance to which the message was sent.

Les types de message-handlers

Handler type	Class Role	Return Value
around	: Set up environment for other handlers	yes
before	: Auxilliary work before primary	no
primary	: Perform major task of message	yes
after	: Auxilliary work after primary	no

- exemple d'usage de handlers de type before et after :

```
CLIPS> (defmessage-handler USER print before ()
    (printout t "*** Starting to print ***" crlf))
CLIPS> (defmessage-handler USER print after ()
    (printout t "*** Finished printing ***" crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
*** Finished printing ***
CLIPS>
```

- a **before handler** type does its task before the **primary**type handler, and an **after handler** does its task after the primary handler.

Predefined USER Message-handlers of Primary Type

- xxx

Primary type	Class Role
init	: Initialize an instance
before	: Delete an instance
primary	: Print the object
after	: Directly modifies slots
before	: Modifies slots using put- messages
primary	: Duplicates instance without put- message
after	: Duplicates an instance using messages

Predefined USER Message-handlers of Primary Type

```
CLIPS> (defmessage-handler USER init before ()
   (printout t "*** Starting to make instance ***" crlf))
CLIPS> (defmessage-handler USER init after ()
   (printout t "*** Finished making instance ***" crlf))
CLIPS> (reset)
*** Starting to make instance ***
*** Finished making instance ***
*** Starting to make instance ***
*** Finished making instance ***
*** Starting to make instance ***
*** Finished making instance ***
CLIPS> (make-instance Dixie_Duck of DUCK (age 1))
*** Starting to make instance ***
*** Finished making instance ***
[Dixie_Duck]
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
[Dorky_Duck] of DUCK
[Dinky_Duck] of DUCKLING
[Dixie_Duck] of DUCK
For a total of 4 instances.
CLIPS>
```


Les fonctions dynamic-get- & dynamic-put-

- used from within a handler to read and write a slot value of the active instance
- exemple :

```
CLIPS> (defmessage-handler USER print-age primary ()
   (printout t "*** Starting to print ***" crlf
               "Age = " (dynamic-get age) crlf
               "*** Finished printing ***" crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

- The ?self:age can only be used in a class and its subclasses which inherit the slot age
- The ?self:<slot-name> is evaluated in a *static* way through inheritance.
- This means that if a subclass redefines a slot, a superclass message-handler will fail if it tries to directly access the slot using ?self:<slot-name>.

Démon

- A **daemon** is a handler which executes whenever some **basic action** like initialization, deletion, get, or put is performed on an instance
- A rule cannot be considered a daemon because it's not certain that it will be executed just because its LHS patterns are satisfied.
- The only thing that is certain is that a rule will become *activated* when its LHS is satisfied — not that it will execute.

declarative implementation : Daemons implemented using **before** and **after handlers** since these will be executed before and after their primary handler

imperative implementation : the actions are explicitly programmed and the **around handler** is very convenient to use for imperative daemons.

The basic idea of the **around handler** is as follows.

1. Start before any other handlers.
2. Call the next handler using either .i.call-next-handler; to pass the same arguments or .i.override-next-handler; to pass different ones.
3. Continue execution when the last handler finishes.
4. After any other around, before, primary, or after handlers finish, the around handler resumes execution

Démon : exemple

- The following example illustrates the around handler through a truthful daemon that tells on Dorky_Duck whenever he lies about his age :

```
CLIPS> (defmessage-handler DUCK lie-about-age around (?change)
   (bind ?old-age ?self:age)
   (if (next-handlerp) then
       (call-next-handler))
   (bind ?new-age ?self:age)
   (if (<> ?old-age ?new-age) then
       (printout t "Dorky_Duck is lying!" crlf
                   "Dorky_Duck is lying!" crlf
                   "He's really " ?old-age crlf)))
CLIPS> (make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

- Although Dorky_Duck may still lie about his age, the daemon tells the truth.
- Notice the ?change argument. Although the around handler does not use ?change, the lie-about-age primary that is called by the call-next-handler does need it to change the age. Thus, ?change must be passed to the primary by the around handler. An error message will occur if you leave out the ?change.
- The call-next-handler *always* passes the arguments of the shadower to the shadowed handler

Autres fonctions utiles avec les handlers

- A number of other functions are useful with handlers :

Function	Meaning
undefmessage-handler	Deletes a specified handler
list-defmessage-handlers	Lists the handlers
delete-instance	Operates on the active instance
message-handler-existp	Returns TRUE if handler exists, else FALSE

- The **grouping functions**group COOL items into a multifield variable :

Function	Meaning
get-defmessage-handler-list	: Groups the class names, message names and types (direct or inherited)
class-superclasses	: Groups all superclass names (direct or inherited.)
class-subclasses	: Groups all subclass names (direct or inherited).
class-slots	: Groups all slot names (explicitly defined or inherited)
slot-existp	: Returns TRUE if class slot exists, else FALSE
slot-facets	: Groups the specified slot facet values of a class
slot-sources	: Groups the slot names of classes which contribute to a slot in the specified class

Autres fonctions utiles avec les handlers

- The **preview-send**function is useful in debugging since it displays the sequence of all handlers that potentially may be involved in processing a message. The reason for the term potentially is that shadowed handlers will not be executed if the shadower does not use call-next-handler or override-next-handler.
- Other functions useful with pattern matching objects by rules follow

Function	Meaning
object-pattern-match-delay	: Delay pattern matching of rules until after instances are created, modified, or deleted
modify-instance	: Modifies instance using slot overrides. Object pattern matching delayed until after modifications.
active-modify-instance	: Change the values of the instance concurrent with object pattern matching with *direct-modify* message
message-modify-instance	: Change the values of the instance. Delay object pattern matching until all slots are changed
active-message-modify-instance	: Change the values of the instance concurrent with object pattern matching using *message-modify*