

III.4 Type concret : description logique

Une fois le type abstrait défini, il faut déterminer *un* codage permettant de l'implémenter. Pour cela, il faut :

- ❑ choisir quelles informations seront réellement codées en tant que valeur ;
- ❑ définir la représentation de ces informations ;
- ❑ définir l'organisation logique des informations ;

III.4 Type concret : description logique

Une fois le type abstrait défini, il faut déterminer *un* codage permettant de l'implémenter. Pour cela, il faut :

- ❑ choisir quelles informations seront réellement codées en tant que valeur ;
- ❑ définir la représentation de ces informations ;
- ❑ définir l'organisation logique des informations ;

Définition : *Description logique*

Établir la *description logique* d'un *type abstrait* consiste à définir la *structure de données* et les algorithmes des fonctions et propriétés de la spécification fonctionnelle.

→ Organisation logique des données à représenter, chaque donnée étant d'un type précédemment défini.

Il existe 5 types de relations entre type de données permettant de structurer l'information et de définir un type complexe :

- ❑ la juxtaposition ;
- ❑ l'énumération ;
- ❑ la collection finie ;
- ❑ la séparation de cas.

III.4.1 La juxtaposition

73

La juxtaposition permet d'agréger les données à stocker

Définition : *Juxtaposition*

```
TypeJuxtapose = Struct(donnée1 : Type1 ; ... ; donnéen : Typen)
```

On peut également l'écrire sous forme extensive :

```
Struct TypeJuxtapose  
  nom_donnée1 : Type1  
  ...  
  nom_donnéen : Typen  
endStruct
```

❑ Exemple :

Pour décrire un modèle de voiture au niveau logique, on peut, par exemple, utiliser la décomposition suivante :

```
Modele = {
  nom:Text
  prixvente: {
    prix:Float
    remiseMax:Float
    remise:Float
  }
  millésime: {
    mois:Int
    année:Int
  }
}
```

❑ Exemple :

Pour décrire un modèle de voiture au niveau logique, on peut, par exemple, utiliser la décomposition suivante :

$$\text{Modele} = \left\{ \begin{array}{l} \text{nom:Text} \\ \text{prixvente:} \left\{ \begin{array}{l} \text{prix:Float} \\ \text{remiseMax:Float} \\ \text{remise:Float} \end{array} \right. \\ \text{millésime:} \left\{ \begin{array}{l} \text{mois:Int} \\ \text{année:Int} \end{array} \right. \end{array} \right.$$

```
Struct Modele = (  
  nom : Text  
  privenite : Struct(prix: Float ; remiseMax: Float ; remise: Float)  
  millésime: Struct(mois: Int ; année: Int)  
)
```

Il est souvent préférable de définir de nouveaux type pour manipuler les données, si le besoin s'en fait sentir.

Ainsi, du seul point de vue de la structure de données (description logique), l'exemple précédent pourrait s'écrire :

```
PrixModele= Struct(prix:Float; remiseMax:Float; remise:Float)

Date = Struct(mois: Int; année: Int)

Modele= Struct(nom:Text; prixvente: PrixModele; millésime:Date)
```

Ou de manière extensive :

```
Struct PrixVente  
  prix : Float  
  remiseMax : Float  
  remise : Float  
endStruct
```

```
Struct Millesime  
  mois : Int  
  année : Int  
endStruct
```

```
Struct Modele  
  nom : Text  
  prixvente : PrixVente  
  millesime : Millesime  
endStruct
```


La juxtaposition dans les langages

La *juxtaposition* s'écrit dans la plupart des langages de programmation comme un *record* ou une *structure*.

En Swift, la juxtaposition se décrit par une *struct* :

Définition : *Struct en Swift*

```
struct Nom {  
    var nom_donné1: Type1  
    ...  
    var nom_donnén: Typem  
};
```

III.4.2 Énumération

78



Une énumération permet de définir un type pour lequel le nombre de valeurs possibles est fini et déterminé

Définition : *Énumération*

```
Enum T(id1 [= val1] ; ... ; idn [= valn])
```

Chaque id_i est un identifiant unique représentant un scalaire. Il peut éventuellement être associé à une valeur réelle

Par défaut, si aucune valeur n'est spécifiée, la valeur est égale au rang. Le premier identifiant prend la valeur 0 et cette valeur est incrémentée pour les identifiants suivants.

Exemple :

```
Enum Mois (janvier,...,decembre)
```

On peut alors redéfinir le type `Date` ainsi

```
Struct Date(mois: Mois; annee: Int)
```

Et on peut vérifier la valeur du mois :

```
func hiver(date: Date) -> Bool{  
    return (date.mois == janvier) || (date.mois == février)  
}
```

```
func mois(date: Date) -> String{  
    switch date.mois{  
    case janvier : return "janvier"  
        ...  
    }  
}
```

L'énumération dans les langages

L'*énumération* reste limité à l'utilisation d'identifiant sans valeur dans la plupart des langages. En Swift elle offre plus de possibilités :

- ❑ Il est possible d'associer une valeur par défaut à un identifiant
- ❑ Il est également possible de spécifier un type différent à chaque identifiant.

Les propriétés *.value* et *.rank* permettent d'obtenir respectivement la valeur et le rang de l'identifiant, sachant que si aucune valeur n'a été spécifiée, ils sont égaux.

Par défaut, si aucun type et aucune valeur n'est spécifiée, la valeur est égale au rang. Le premier identifiant prend la valeur 0 et cette valeur est incrémentée pour les identifiants suivants.

Définition : *Énumération en Swift*

```
enum NomEnum {  
    case id1  
    case id2 = valeur  
    case id3(String)  
    case id4(Int) = 5  
    ...  
}
```

Pour vérifier une valeur on utilisera l'instruction switch associé à let :

```
Switch nom{  
case id3(let s): ...  
}
```

Exemple :

```
enum CodeId{
    case barcode(Int,Int,Int,Int) = (0,0,0,0)
    case qrCode(String)
}

func printCode(codeproduit: CodeId){
    switch codeproduit{
    case CodeId.qrCode(let code):
        print("Qrcode = \(code)")
    case .barcode(let system, let usine, let produit, let verif):
        print("system = \(system)")
    }
}
```

III.4.3 Collection finie : les tableaux

Structure logique permettant de représenter des collections de valeurs d'un type fixe.

Une collection finie a un nom fixe, un type fixe et un nombre fixe de variables qui lui sont associées.

Définition : *Collection finie*

$$T = [Type_1] (n)$$

Chaque variable est formée du nom de la collection et d'un indice.

Notation : la i ème valeur de la collection C de type T s'évalue grâce à l'expression suivante : $C[i-1]$

Les tableaux sont des types permettant de regrouper un nombre N de données du même type sous la même entité (dans la même variable). Ce nombre N est généralement fixé et déterminé à la déclaration du tableau dans le programme.

On parle alors de *tableaux statiques*, appelés par défaut *tableaux*.

Certains langages offrent la possibilité de définir des tableaux dont la taille n'est connue qu'à l'exécution, voir de tableaux dont la taille peut varier.

On parle alors de *tableaux dynamiques*.

Un tableau est donc un type de données permettant à une variable de type *tableau* de stocker un nombre N , connu à l'avance et **fixé dans le programme**, de données du **même type** dans les cases du tableau.

Un tableau possède uniquement deux opérations :

- affectation d'un élément dans une case du tableau
- lecture d'un élément dans une case du tableau

et une propriété : la taille du tableau

Le schéma de définition d'un tableau est le suivant :

```
nom_tableau : [Type](N) = val
```

Toutes les données stockées dans le tableau sont de type **Type** et le tableau stocke exactement N données qui ont toute pour valeur initiale `val`.

Pour lire la valeur d'une donnée, on utilise l'opération `[]`.

Cet opérateur prend en argument la position la position dans le tableau de la donnée que l'on veut lire. La déclaration suivante

```
nom_tableau[i]
```

permet d'accéder à la $(i+1)$ ème donnée du tableau, celle-ci étant de type `T`. En effet, les cases du tableau sont numérotées à partir de 0.

Ainsi

```
var x : Type = nom_tableau[0]
```

permet de récupérer dans `x` la valeur de la 1ère case du tableau.

```
nom_tableau[0] = x
```

permet de changer la valeur de la 1ère case du tableau.

Les tableaux dans les langages

Les *tableaux* sont la première *structure de données* que on va étudié. Précisons sa représentation physique.

En Swift, le type collection privilégié est le tableau. Mais c'est un tableau dynamique qui s'apparente donc dans son fonctionnement à une liste.

Les tableaux swift ne sont donc pas des tableaux au sens algorithmique du terme puisque leur taille peut varier, mais pas non plus des listes séquentielles puisqu'on peut accéder directement à un élément par son index.

Dans un premier temps, on restreindra l'utilisation des tableaux swift à une utilisation d'un tableau algorithmique, en restreignant donc les opérations autorisées sur ceux-ci

Les tableaux/listes en swift sont un type valeur :

- ❑ l'opération d'affectation copie donc les tableaux
- ❑ il ne faut pas confondre *indice* et valeur du tableau :

```
tab1[i] ≠ i
```

- ❑ l'indice d'un tableau peut être n'importe quelle expression évaluable dont la valeur est de type *Int* ;
- ❑ chaque élément du tableau se comporte comme une variable, il faut donc *initialiser* sa valeur avant de pouvoir l'utiliser ;
- ❑ la fonction ou la propriété *count* permettent de connaître la taille d'un tableau ; par exemple

```
tab1.count == n
```

Exemple de déclaration de tableaux :

Soit le tableau suivant : [3, 6, 88, 27, 54]

Celui-ci peut avoir été déclaré et initialisé par les séquences d'instructions suivantes :

```
var T1 = [Int](repeating: 0, count: 5)
T1[0]=3; T1[1]=6; T1[2]=88; T1[3]=27; T1[4]=54;
ou
var T2 : [Int] = [3, 6, 88, 27, 54]
```

Exemple d'utilisation de tableaux

L'opération « *l'élément de tableau précédant l'élément i reçoit la valeur de l'élément i* » s'écrit :

```
T1[i-1] = T1[i]
```

L'opération « *l'élément de tableau suivant l'élément i reçoit la valeur de l'élément i* » s'écrit :

```
T1[i+1] = T1[i]
```

Les itérations inconditionnelles sont le moyen naturel de parcourir un tableau :

```
var T1 : [Int](repeating: 0, count: 5)
for i in 0..<5 do
    T1[i]=i+1
endfor
```

En résumé, on utilisera une structure de donnée *tableau* si :

- ❑ le nombre maximal de données est déterminé à l'avance ;
- ❑ les données ont besoin (peuvent) être indexées ;
- ❑ on peut accéder aux données par leur indice ;
- ❑ les données sont toutes du même type.

Un tableau peut être multi-dimensionnel :

```
M1 = [[Int]] = [[00,01,02],[10,11,12]]  
M2 = [[[Int]]] = [  
    [[000,001,002,003],[010,011,012,013]], [[100,101,102,103],  
    [110,111,112,113]], [[200,201,202,203],[210,211,212,213]]  
]
```

III.4.4 Alternative ou Séparation de cas

92



Alternative pour le choix du type :

Lorsque pour une structure de données, deux types sont possibles, une solution est alors de définir deux types possibles pour une même donnée :

Définition : Séparation de cas

$T = (\text{Type}_1 \mid \text{Type}_2)$

L'opérateur logique `isKindOf(v, T)` permet de vérifier le type d'une variable.

La séparation de cas dans les langages

Les *séparation de cas* et la *récurtivité* peuvent rarement être traduite directement dans les langages de programmation. À ma connaissance, seul *TypeScript* permet de le faire.

Les langages *Kotlin* et *Swift* offrent aussi cette possibilité mais de manière détournée :

- ❑ l'utilisation de *types optionnels* : pour l'alternative avec *Vide*
- ❑ les *énumérations* qui sont de véritables types

Dans les autres langages, comme le C, il faudra utiliser un palliatif pour simuler la séparation de cas en utilisant des *références* à des objets, quand c'est possible.

III.4.4.1 La séparation de cas : références

Dans la plupart des langages, on simulera la séparation de cas grâce aux *références (pointeurs)* qui permettent de référencer des objets (au sens générique du terme). Ils seront utilisés pour coder l'alternative : *(Vide | Type)*

Définition : *Référence*

Les pointeurs sont typés et leur évaluation est une adresse (*référence*) indiquant l'emplacement d'une valeur du type du pointeur.

```
typeref (x1, ... , xn) Type
```

La syntaxe **T* permet d'indiquer que l'on veut évaluer la valeur en mémoire à l'adresse stockée dans la variable pointeur.

Le type pointeur, ou référence, possède une valeur particulière permettant d'indiquer que rien n'est référencé par la variable pointeur : la valeur *nil* (ou *Null* ou tout autre variante).

Pour coder une alternative de type `(ListeNV|Vide)` on utilisera donc un pointeur de type `*Liste`, et le type `Vide` sera alors *codé par la valeur nil*.

On s'attachera donc à tester si la valeur du pointeur est nil pour vérifier si l'on est dans le cas `Vide` ou pas.

Remarque : les références aux objets permettent d'implémenter d'autres types d'alternatives grâce à la notion d'héritage.

Exemple : implémentation d'une alternative

En supposant que l'on dispose d'un type référence `TypeR` (par exemple `Ship` pour la bataille navale), ce type code en fait l'alternative (`TypeR|Vide`) puisqu'une variable de type `TypeR` pourra avoir la valeur `nil`, c'est à dire `Vide` :

À l'utilisation, il faudra vérifier à chaque fois qu'une variable de type `TypeR` n'est pas `Vide` au risque sinon que le programme soit interrompu par une erreur. :

```
func changeTaille(b: Ship){  
    if b == nil then  
        // pas de bateau par ici  
    else  
        // on a bien un bateau, on peut changer sa taille par exemple  
        b.taille = 3 // sans vérification, on prenait le risque que b  
                    // soit Vide et alors provoque une erreur (Null Pointer Exception  
                    // ou segmentation fault)  
    endif  
}
```

III.4.4.2 La séparation de cas : enum

Les énumérations (type valeur) couplées au `switch case`, permettent de gérer des alternatives ainsi le type voiture avec une remise sous forme d'entier ou de pourcentage pourrait être :

```
enum Remise{  
    case Valeur(Int)  
    case Pourcentage(Float)  
}
```

III.4.4.2 La séparation de cas : enum

97



Les énumérations (type valeur) couplées au `switch case`, permettent de gérer des alternatives ainsi le type voiture avec une remise sous forme d'entier ou de pourcentage pourrait être :

```
struct Voiture{
    init(nom: String, remise_max: Remise, prix: Float){
        self.nom = nom ;self.remise = remise_max; self.prix = prix
    }
    private(set) var nom : String
    public var a_remise : Bool {
        switch self.remise{
            case .Valeur(let val): return val > 0
            case .Pourcentage(let p): return p > 0
        }
    }
    public var remise : Remise
    private(set) var prix: Float
    public func prix(remise: Float) -> Float{
        switch self.remise {
            case .Valeur(let val): return self.prix - Float(val)
            case .Pourcentage(let p): return (1-p)*self.prix
        }
    }
}
```

```
enum Remise{
    case Valeur(Int)
    case Pourcentage(Float)
}
```

III.4.4.3 La séparation de cas : optionnels

Les types optionnels seront utilisés pour indiquer si une variable peut ne pas avoir de valeur : on est dans le cas d'une alternative (Type valeur | Vide) ; par exemple :

```
var v : Ship? // équivalent à v : (Ship | Vide)
```

L'avantage de cette solution est qu'elle permet un contrôle strict de l'alternative (Type | Vide) et de spécifier si une valeur est requise ou alors si on accepte Vide. L'autre avantage est qu'un type optionnel, peut être un type valeur. Ainsi, par exemple :

```
func changeTaille(b: Ship, val: Int){  
    b.taille = val // nul besoin de vérification, b est de type  
    Ship et non pas (Ship | Vide) : pas de risque d'erreur  
}  
var v : Int? = 3    // possible, 3 est un Int, v est aussi un Int  
var w : Int? = nil  // possible, w peut aussi être Vide  
var i : Int = v     // impossible, v pourrait être vide, i ne le  
                    // peut pas
```

III.4.4.3 La séparation de cas : optionnels


98



Les types optionnels seront utilisés pour indiquer si une variable peut ne pas avoir de valeur : on est dans le cas d'une alternative (Type valeur | Vide) ; par exemple :

```
var v : Ship? // équivalent à v : (Ship | Vide)
```

L'avantage de cette solution est qu'elle permet un contrôle strict de l'alternative (Type | Vide) et de spécifier si une valeur est requise ou alors si on accepte Vide. L'autre avantage est qu'un type optionnel, peut être un type valeur. Ainsi, par exemple :

```
func changeTaille(b: Ship, val: Int){  
    b.taille = val // nul besoin de vérification, b est de type  
    Ship et non pas (Ship | Vide) : pas de risque d'erreur  
}  
var v : Int? = 3    // possible, 3 est un Int, v est aussi un Int  
var w : Int? = nil  // possible, w peut aussi être Vide  
var i : Int =  // impossible, v pourrait être vide, i ne le  
// peut pas
```


III.4.5 Définir le codage d'un type

Une fois la structure de donnée choisie, la dernière étape consiste à choisir comment coder le type et à écrire les fonctions.

- définir le *codage* des données et des algorithmes :
- ❑ choix du codage du type (type valeur ou type référence)
- ❑ *implantation* des algorithmes dans un langage de programmation

III.4.5.1 Type valeur en Swift : Struct

Pour définir un types valeur, Swift privilégie les `structs`.

Les `structs` permettent à la fois :

- ❑ de définir concrètement un *type valeur*
- ❑ d'organiser les données par *juxtaposition*
- ❑ d'intégrer (*encapsuler*) l'implémentation des fonctions de la spécification :
 - données « protégées » au sein de l'espace de dénomination de l'objet ;
 - propriétés et fonctions (appelées méthodes) « protégées » au sein de l'espace de dénomination de l'objet.

```
protocol NomProtocol{  
    var data : type { get }  
    init(paramètres)  
    f(paramètres) -> TypeRetour  
}
```



```
protocol NomProtocol{
    var data : type { get }
    init(paramètres)
    f(paramètres) -> TypeRetour
}
```

L'implémentation de ce type se fera de la manière suivante :

Définition : *Struct en Swift*

```
struct NomType : NomProtocol {
    private(set) var data : type // propriété du type

    init(paramètres){ // fonction d'initialisation/création
        // définition d'un attribut d'instance
        self.data = type(valeur)

    }

    func f(paramètres) -> TypeRetour { // fonction du type abstrait
        <expressions>
        return value
    }
}
```

Comment utilise-t-on un type ainsi défini ?

- ❑ création : comme une définition classique :

```
x = Type(param1,param2)
```

- ❑ appel aux fonctions :

```
x.f(...)
```

- ❑ accès aux propriétés (*attention* accesseurs définis par défaut):

```
x.v
```

Peut-on masquer l'implémentation ?

oui en utilisant `private...` utile uniquement pour les fonctions internes, les propriétés et les fonctions de la spécification doivent rester publiques

III.4.5.2 Type valeur en Swift : Enum

Les Enum en swift permettent également de définir des types.

Les `enum` permettent à la fois :

- ❑ de définir concrètement un *type valeur*
- ❑ d'organiser les données par *énumération*
- ❑ d'intégrer (*encapsuler*) l'implémentation des fonctions de la spécification :
 - données « protégées » au sein de l'espace de dénomination de l'objet ;
 - propriétés et fonctions (appelées méthodes) « protégées » au sein de l'espace de dénomination de l'objet.

Exemple :

```
enum Tenum{  
    case val(Int)  
    case pou(Float)  
    func hello(){  
        print("coucou")  
    }  
}  
  
var v = Tenum.val(3)  
v.hello()
```

III.4.5.3 Type référence en Swift : Class

Les Class en swift permettent de définir des types référence.

Les `class` permettent à la fois :

- ❑ de définir concrètement un *type référence*
- ❑ d'organiser les données par *juxtaposition*
- ❑ d'intégrer (*encapsuler*) l'implémentation des fonctions de la spécification :
 - données « protégées » au sein de l'espace de dénomination de l'objet ;
 - propriétés et fonctions (appelées méthodes) « protégées » au sein de l'espace de dénomination de l'objet.
- ❑ D'autres mécanisme sont en œuvre que vous verrez au second semestre.

III.4.6 Description logique : Conclusion

Une *description logique* d'un type n'est complète que si on rajoute à la définition de la structure de données la description des opérations.

Il faut donc, pour chaque fonction définie dans la spécification fonctionnelle, définir l'algorithme permettant de réaliser l'opération.

Ces algorithmes devront en outre vérifier les propriétés énoncées dans la spécification fonctionnelle.

Définition : *Description logique : Type concret*

Une *description logique* d'un type, dont la *spécification fonctionnelle* est connue, comprend :

- ❑ La définition d'une *structure de données* décrivant la structuration logique des informations par *juxtaposition*, *séparation de cas*, *énumération*, ou *collection finie*, en faisant intervenir des types définis auparavant et éventuellement le type lui-même (*récurtivité*). En cas de récursivité, on utilisera le type spécial *Vide* pour marquer la fin de la récursivité.
- ❑ La définition des *algorithmes des fonctions* opérant sur les éléments du type définis dans la structure de données. À chaque opération de la description fonctionnelle doit être associé un de ces algorithmes, vérifiant les caractéristiques de la fonction. Si un composant de la structure de données est défini par séparation de cas, il faut utiliser l'opération *isKindOf* pour vérifier le type exacte de la donnée afin d'accéder correctement à ses propriétés et fonctions.