

Structures de Données

Type Abstraits de Données

Contenu

- Types Abstraits de Données
 - Définition
 - Implémentation
 - Encapsulation
 - Interface

Types Abstraits de Données

Motivation

- Formaliser la définition d'un nouveau type
 - Complément aux outils de modélisation déjà vus
- Permet une description en langage courant mais rigoureuse
 - Compacte
 - Exhaustive
- Séparer la modélisation de l'implémentation
 - Phase initiale de réflexion sur les fonctionnalités
 - Transfert vers une mise en place
- Simplifier la communication
 - Outil standard, clair
 - « auto-documenté »

Pourquoi créer un nouveau type?

- Répondre de façon formelle à un problème
 - Créer un **outil** pour proposer une solution
 - Avoir un outil structuré pour **tester sa solution**
- ⇒ Premier pas vers le Génie Logiciel
- ⇒ Modélisation de processus
 - ⇒ Modélisation de logiciels

A noter: notion initiale vers l'**objet** (au sens informatique)

Exemple

Problème: Gestion de ses **contacts**

Solution: Création d'un **carnet** d'**adresses**

Modélisation: Liste de **personnes**

⇒ Création d'un type de données « **personne** »

⇒ Mise en place d'une liste de **personne**

Le type **TPersonne**

⇒ Stocke des données (nom, prénom, adresse, tel,...)

⇒ Est associé à des opérations (modification,...)

⇒ Une fois créé, ce type pourra être utilisé comme un type classique

⇒ On crée un **outil réutilisable**

Méthodologie


1. Compréhension du problème posé
 2. Modélisation
 3. Fiche technique du TAD
 4. Implémentation / Documentation
- ⇒ La définition du type se situe juste après l'analyse du problème et débute la définition d'une solution formelle (implémentée)

Modélisation

Choix des TADs :

- Un nouveau type doit représenter un concept majeur du problème
 - Identifier les concepts (nom communs, ...)
 - Rassembler ces concepts (synonymes, hierarchie,...)

Selection:

- Un TAD est lié aux objectifs du problème
- Un TAD rassemble un contenu et des opérations
 - On assure la coherence avec la sémantique
- Un TAD permet de faire une "liste de" ce type
 - "pointeur sur"
- L'utilisation d'un TAD simplifie le code et le clarifie
 - Bénéfices en termes de documentation

Dans notre exemple...

- La **personne** est identifiée comme l'élément clé de la solution
- On doit définir ce qu'est une personne (au sens du carnet d'adresse)
 - Une personne a un nom, prénom, adresse,...
 - Une personne déménage, s'achète un Natel, décède...
- On doit passer de cette notion de personne à un objet informatique de type **TPersonne**
 - Propriétés de la personne
 - Opérations associées

A noter: on pourrait définir un type **TCarnetDAdresse**

Type Abstrait de Données (TAD)

- On crée une **fiche technique** du nouveau type
 - Présentation standard

- On spécifie (en langage courant):
 - Le but du TAD
 - Comment le construire (ses prérequis, dépendances, attributs)
 - A quelles opérations spécifiques est associé notre TAD

- On définit
 - Les opérations utiles à son utilisation
 - Les opérations internes

- Le but est de proposer un type **simple mais complet**

Notation

Nom du TAD**Rôle, but:**

→ Décrire de façon concise le nouveau type de données

Spécification, déclaration:

→ Préciser les dépendances et contraintes éventuellement nécessaires à son élaboration

Primitives, opérations:**1. Création, initialisation, construction**

Lister les opérations servant à l'initialisation des instances du type

2. Modification, transformation

Lister les opérations permettant la modification d'une instance du type (insertion, suppression,...)

3. Destruction

Lister les opérations permettant la destruction d'une instance du type

4. Accès, observation

Lister les modes d'accès à une instance du type (parcours,...)

Formalisme classique

Pour décrire un type abstrait on peut aussi parler de:

- Dépendance: de quel autre type abstrait dépend ce type

- Exemple: TEmploye dépend de TPersonne

Ces dépendances sont exprimées dans les **diagrammes** (lien, flèches)

- Axiomes et pré-conditions: ce sont les contraintes qui assurent le bon déroulement des opérations de construction, modification, accès, destruction

- Exemple: Diviser par zéro, accéder à un élément d'une liste vide

Ici, on exprimera ces cas par des **exceptions**

Utilité du TAD

- Le TAD permet de faire la transition vers une représentation formelle (informatique)
- Il permet de construire la définition formelle de manière itérative
 1. On **défini les besoins** de manière **informelle**
 2. On **traduit ces besoins** de façon **formelle**
 3. On **teste l'efficacité** du type créé
 4. Si besoin, on recommence le cycle
- A la fin de la spécification, il reste un descriptif que l'on peut transmettre
 - Pour faire effectuer l'implémentation
 - Pour documenter son utilisation

De la spec. à l'implémentation

Spécification

Nom du TAD

Rôle, but:

→ Décrire de façon concise le nouveau type de données

Spécification, déclaration:

→ Préciser les contraintes éventuellement nécessaires à son élaboration

Primitives, opérations:

1. Création, initialisation, construction

Lister les opérations servant à l'initialisation des instances du type

2. Modification, transformation

Lister les opérations permettant la modification d'une instance du type (insertion, suppression,...)

3. Destruction

Lister les opérations permettant la destruction d'une instance du type

4. Accès, observation

Lister les modes d'accès à une instance du type (parcours,...)

Implementation

```
/* Type TMonType: permet de ...
```

```
... commentaire ...
```

```
*/
```

```
type
```

```
TMonType : ... description ...
```

```
procedure init()
```

```
procedure insere()
```

```
procedure ferme()
```

```
fonction TMonType trouve()
```

Type Abstrait de Données (TAD)

- Est une spécification
 - Ne détaille que ce qui est nécessaire à la complétude du type (**minimaliste**)
 - Décrit ce qui est nécessaire à un contrôle total et utile de la structure (**exhaustif**)
- ⇒ Pas de détails d'implémentation
Ces détails ne seront pas accessibles à l'utilisateur de la structure
- ⇒ Indépendant du langage
Peut être implémenté dans différents langages de manières différentes en fonction des propriétés de base de chaque langage

Exemple

Création du type « TPersonne »

Une personne est définie par:

- Son nom, prénom, genre
- Son age, sa date de naissance
- Son adresse, ses numéros de téléphone

- Elle peut être créée, effacée
- Elle peut être modifiée
 - Nom, adresse, numéros de téléphone
- On peut lui demander
 - Son nom, adresse, ses numéros de téléphone

A noter: ces listes peuvent être complétées plus tard

Type Personne

Nom du TAD

Rôle, but:

→ Décrire de façon concise le nouveau type de données

Spécification, déclaration:

→ Préciser les contraintes éventuellement nécessaires à son élaboration

Primitives, opérations:

1. **Création, initialisation, construction**
Lister les opérations servant à l'initialisation des instances du type
2. **Modification**
Lister les opérations permettant la modification d'une instance du type (insertion, suppression,...)
3. **Destruction**
Lister les opérations permettant la destruction d'une instance du type
4. **Accès**
Lister les modes d'accès à une instance du type (parcours,...)

Personne

→ Permet de stocker les données correspondant à une personne

Spécification, déclaration:

→ Description d'une adresse, téléphone

Création:

- a. Spécifier au moins le nom (*et prénom?*)

Modification:

- a. Changer le nom
- b. Changer un numéro de téléphone
- c. Ajouter un numéro de téléphone
- d. Supprimer un numéro de téléphone
- e. Changer l'adresse

Destruction:

- a. Libération de l'espace

Accès:

- a. Donne le nom
- b. Donne l'adresse
- c. Donne les numéros de téléphone
- d. ~~Donne le nombre de numéros de téléphone?~~
- e. ~~Donne un résumé?~~



Pas vers l'implémentation

Personne

→ Permet de stocker les données correspondant à une personne

Spécification, déclaration:

→ Description d'une adresse, téléphone

Création:

- Spécifier au moins le nom (*et prénom?*)

Modification:

- Changer le nom
- Changer un numéro de téléphone
- Ajouter un numéro de téléphone
- Supprimer un numéro de téléphone
- Changer l'adresse

Destruction:

- Libération de l'espace

Accès:

- Afficher le nom
- Afficher l'adresse
- Afficher les numéros de téléphone
- Afficher le nombre de numéros de téléphone?*
- Afficher un résumé?*

type TPersonne

init (personne, nom, prenom)

changeNom (personne, nouveauNom)

changeTel (personne,
ancienTel, nouveauTel)

ajouteTel (personne, numero)

effaceTel (personne, ancienTel)

changeAdresse (personne,
nouvelleAdresse)

detrui (personne)

donneNom (personne)

donneAdresse (personne)

donneNumeros (personne)

Details d'implémentation

```

type TPersonne

init(personne,nom,prenom)
changeNom(personne, nouveauNom)
changeTel(personne,
           ancienTel,nouveauTel)
ajouteTel(personne,numero)
effaceTel(personne,ancienTel)
changeAdresse(personne,
              nouvelleAdresse)
detruit(personne)
donneNom(personne)
donneAdresse(personne)
donneNumeros(personne)

```

```

/* Type TPersonne: permet de ...
   ... commentaire ...*/

```

```
type
```

```
TPersonne : ... description ...
```

```
procedure init()
```

```
fonction TPersonne changeNom()
```

```
procedure detruit()
```

```
fonction TPersonne changeTel()
```

```
trouveTel(...)
```

```

/* Détails d'implémentation
   spécifiques au langage ou à la
   gestion précise de la structure */

```

```
entier : compteur
```

```
procedure trouveTel(...)
```

Fonctions « internes »

Dans l'exemple précédent la fonction `trouveTel()` est nécessaire pour chercher l'index du numéro de téléphone à remplacer

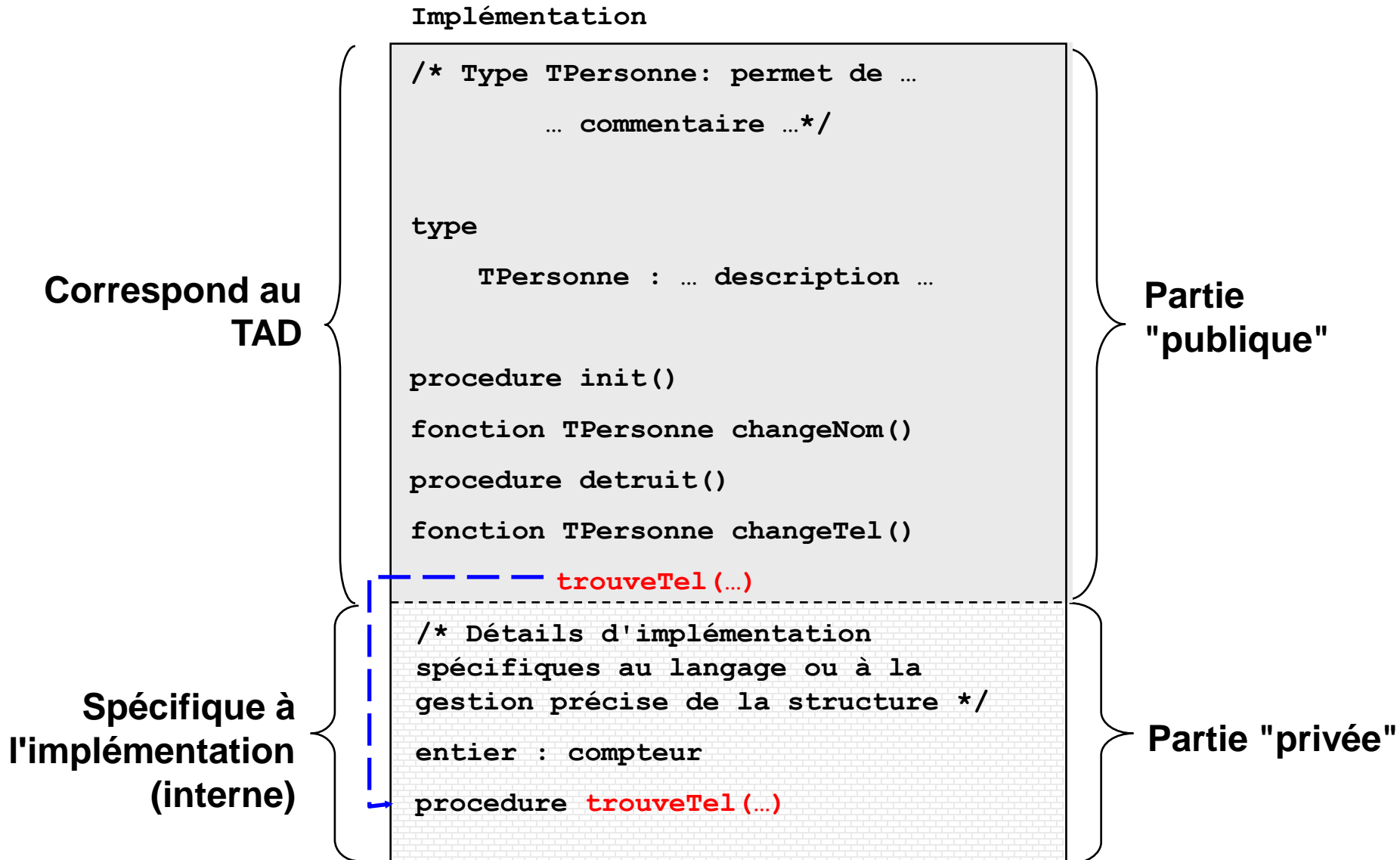
Cet index est un paramètre interne. Cette fonction n'a pas de raison d'être proposée comme fonctionnalité attachée au type, elle est "privée"

Une autre implémentation pourrait utiliser une autre stratégie pour la même fonctionnalité

Cette fonction « privée » serait aussi utile dans la fonction « publique » `effaceTel()`

Ceci justifie le fait d'en faire une fonction (et éviter de dupliquer le code)

TAD vs Implémentation



Variation d'implémentation

- Le TAD correspond à un modèle bâti à partir d'une certaine sémantique
- ⇒ Son utilisation n'a pas de raison de varier en fonction de la manière de l'implémenter
- ⇒ Pour deux implémentations, un même TAD offrira la même partie publique
 - ⇒ Des noms de fonctions standardisés
`read, write, push, pop, ...`
 - ⇒ *Des noms de variables standardisés*

On va plus loin en utilisant la technique
d'encapsulation des variables
(qui rajoute une motivation à la création d'un type)

Encapsulation des variables

On veut rester proche de la sémantique au maximum

- ⇒ On va surtout décrire les **actions** (procédures, fonctions) et pas l'état interne (variables)
- ⇒ On veut cacher les variables internes (**attributs**) afin de garder une flexibilité dans la manière d'implémenter
- ⇒ On réalise **l'encapsulation** des variables en donnant accès à ces variables uniquement par des fonctions
- ⇒ On peut en plus garder un **contrôle** sur les opérations faites sur ces variables

Pas d'encapsulation

```
/* type TPersonne: permet de stocker les données relatives a une personne
physique et de les imprimer dans un format standard */
```

```
type
```

```
TPersonne : struct
```

```
    car[20]: nom
```

```
    car[20]: prenom
```

```
    entier[N_MAX_TEL]: numeroDeTelephone
```

```
    entier: anneeDeNaissance // annee de naissance
```

```
    entier: age
```

```
procedure init(TPersonne laPersonne, car[] nom, car[] prenom)
```

```
    laPersonne.nom <- nom
```

```
    laPersonne.prenom <- prenom
```

Implémentation
du TAD

```
TPersonne: personne // utilisation du TAD
```

```
init(personne, "Einstein", "Albert")
```

```
personne.anneeDeNaissance <- 1879 // on accède directement a la variable
```

```
personne.age <- 25 // source d'erreur possible
```

```
imprime(personne) // fonction externe au TAD
```

Encapsulation sans contrôle

```
/* type TPersonne: permet de stocker les données relatives a une personne physique*/
```

```
type
```

```
TPersonne : struct
```

```
    car[20]: nom, prenom
```

```
    entier: anneeDeNaissance, age
```

```
fonction entier donneAge(TPersonne unePersonne)
```

```
    retourne unePersonne.age
```

```
procedure definirAge(unePersonne, unAge)
```

```
    unePersonne.age <- unAge
```

```
procedure init(TPersonne laPersonne, ...)
```

```
    ... // procédure adaptée a l'encapsulation
```

```
TPersonne: personne // utilisation du TAD
```

```
definirNom(personne, "Einstein")
```

```
definirPrenom(personne, "Albert")
```

```
definirAnneeDeNaissance(personne, 1879)
```

```
definirAge(personne, 25) //toujours l'erreur
```

Encapsulation:
pour chaque variable:
definir et donne

Implémentation
du TAD

Encapsulation avec contrôle

```
/* type TPersonne: permet de stocker les données relatives a une personne physique */
```

```
type
```

```
TPersonne : struct
```

```
    car[20]: nom, prenom
```

```
    entier: anneeDeNaissance // on supprime l'age !
```

```
fonction entier donneAge(TPersonne unePersonne)
```

```
    retourne ANNEE_COURANTE - unePersonne.anneeDeNaissance
```

```
procedure definirAnneeDeNaissance (unePersonne, unAnDeNaissance)
```

```
// On peut avoir un contrôle sur l'anDeNais eventuellement
```

```
    personne.anneeDeNaissance <- unAnDeNaissance
```

```
procedure init(TPersonne laPersonne, ...)
```

```
    ... // cette procedure ne change pas!
```

```
TPersonne: personne // utilisation du TAD
```

```
definirAnneeDeNaissance(personne, 1879)
```

```
// definirAge(personne, 25) : inutile donc pas d'erreur
```

Implémentation
du TAD

Encapsulation

L'encapsulation permet de rester sur une définition sémantique de la structure:

- **definirAnneeDeNaissance** (personne,...) plutôt que `personne.anneeDeNaissance <- ...`

L'encapsulation permet de gérer de façon cohérente des données attachées à la structure

- **age** et **anneeDeNaissance** seront toujours compatibles entre eux dans l'exemple précédent
- on peut ajouter un contrôle pour détecter les cas impossibles ou improbables:

anneeDeNaissance > ANNEE_COURANTE

Encapsulation

L'encapsulation permet de ne pas introduire des artifices dépendant de l'implémentation

➤ On ne doit pas définir **age** et **anneeDeNaissance** mais on laisse l'accès aux deux valeurs.

➤ **donneAnneeDeNaissance**

➤ **donneAge**

⇒ L'age est bien vu comme une conséquence de la valeur de l'année de naissance

Impact sur le TAD

- L'encapsulation permet de ne mettre en avant que les actions faites avec le nouveau type, la manipulation de ses variable étant faite par des procédures d'affectation (**definir**) et des fonctions de récupération (**donne**)
- La partie publique peut donc ne contenir que des définitions de fonctions et de procédures (pas de variables)
- L'implémentation étant elle-même privée, l'exposition de l'implémentation du TAD consistera donc simplement en une liste d'entêtes (**prototypes**) de procédures et fonctions réalisant la sémantique du TAD

⇒ On parle d'**interface**

Interface

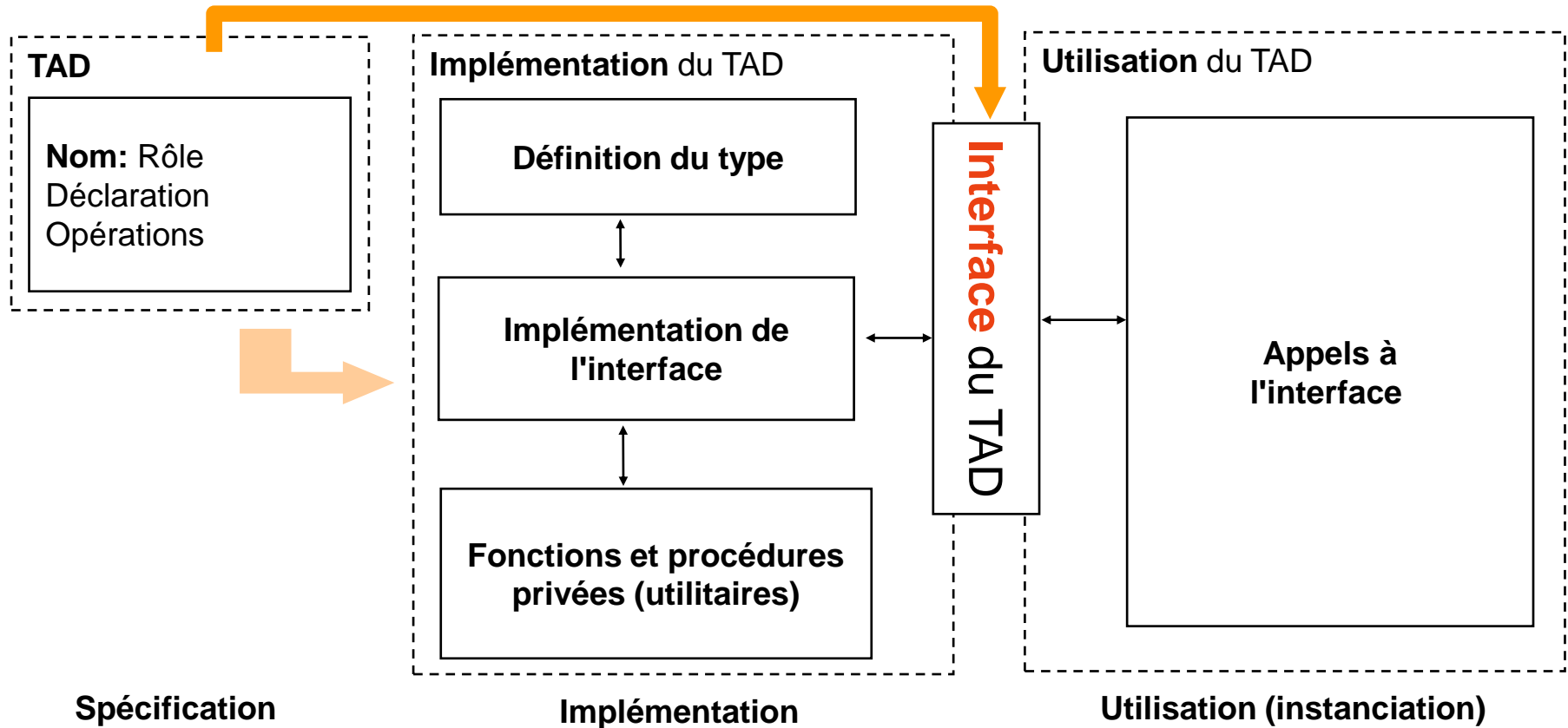
- *Limite commune à deux ensembles ou appareils*
- *Jonction permettant un transfert d'information entre deux éléments d'un système informatique*
 - ⇒ *Connexion, liaison*

Ici:

- On veut définir la partie visible d'un TAD
 - C'est la connexion entre le TAD et son utilisation
 - C'est aussi le "mode d'emploi" du TAD
 - L'interface "cache" les détails d'implémentation
-
- On parle souvent d'API (*Application Programming Interface*)

A noter: Interface graphique (ne pas confondre la terminologie):
⇒ fait partie des Interfaces Homme-Machine (IHM)

Principe de l'interface



Interface : exemple

Implémentation

```
/* Type TPersonne: permet de ...
   ... commentaire ...*/
```

```
type
```

```
    TPersonne : ... description ...
```

```
procedure init()
```

```
    ...
```

```
fonction TPersonne changeNom()
```

```
procedure detruit()
```

```
fonction TPersonne changeTel()
```

```
    trouveTel(...)
```

```
/* Détails d'implémentation
   spécifiques au langage ou à la
   gestion précise de la structure */
```

```
entier : compteur
```

```
procedure trouveTel(...)
```

```
/* Type TPersonne: permet de ...
   ... commentaire ...*/
```

```
type
```

```
    TPersonne : ... description ...
```

```
/* Initialisation
```

```
Parametres:
```

```
    car[] nom : Nom de la personne
```

```
    car[] prenom: Prenom de la personne
```

```
Permet d'initialiser la structure
```

```
*/
```

```
procedure init(nom,prenom)
```

```
/* Changement du nom
```

```
Parametres:
```

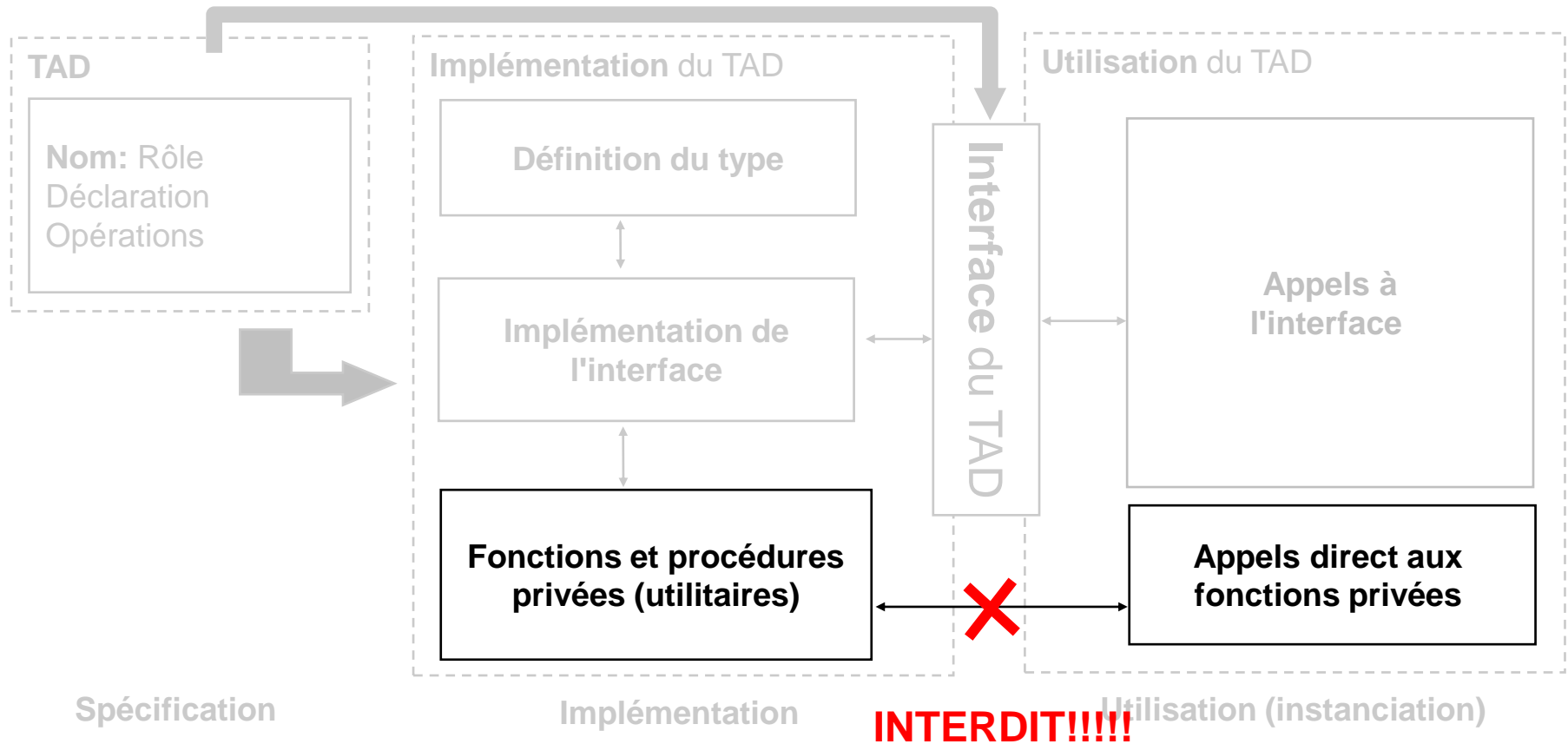
```
    TPersonne: Personne: ... */
```

```
fonction changeNom(...)...
```

Interface
(entêtes)

Partie "privée"

Principe de l'interface



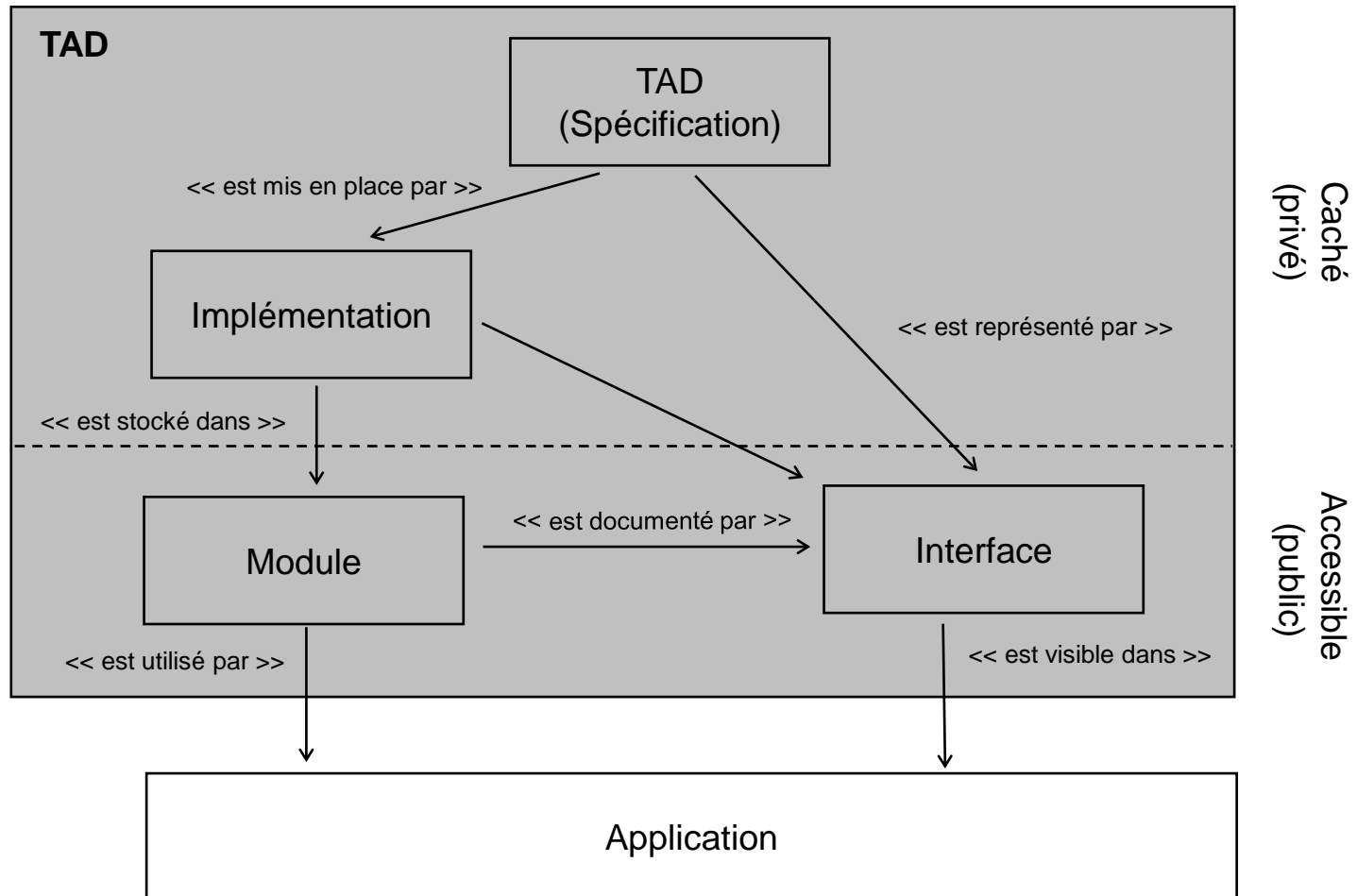
Certains langages proposent un mécanisme de protection de l'interface (**private**)

- C++
- Java
- ...

Notion de module

- Afin de concrétiser la structure de l'interface, on crée un module:
 - Une implémentation privée est créée
 - La seule partie visible est l'interface dans un fichier séparé
- C, C++
 - Le fichier d'implémentation (`.c` , `.cpp`) contient le code
 - Le fichier d'entête (le header, `.h`, `.hpp`) contient les entêtes (prototypes) des fonctions de l'interface
 - Le programme utilisant l'interface inclus le fichier d'entête
- Pascal
 - Le fichier d'implémentation crée une unité (`unit`)
 - Les mots-clés `interface` et `implementation` séparent les parties privées et publiques
 - Le programme inclus le fichier grâce à `uses`

Structure modulaire



Exemple (Pascal)

```

Unit Nom      {Nom de l'unité}

Interface      {Partie "publique"}
  Uses Unité1, Unité2;
  Const Const1 : Type = Valeur;
        Const2 : Type = Valeur;
  Types Type1
        Type2
  Var Var1 : Type;
      Var2 : Type;
  -- Procedure procedure1(arg1,arg2);      {Déclarations des procédures & fonctions}
  -- Function function1(arg1,arg2) : Type; {pouvant être utilisées par les programmes}

Implementation {Partie "privée"}
  Uses Unité1, Unité2;
  Const Const1 : Type = Valeur;
        Const2 : Type = Valeur;
  Types Type1
        Type2
  Var Var1 : Type;
      Var2 : Type;
  -- Procedure procedure1(arg1,arg2);
  Var pVar1 : Type;
      pVar2 : Type;
  Begin
    [Instructions]
  End;
  -- Function function1(arg1,arg2) : Type;
  Var pVar1 : Type;
      pVar2 : Type;
  Begin
    [Instructions]
  End;

```

Exemple (C)

```
/* Structure point */
```

```
typedef struct
```

```
{
```

```
    int x;
```

```
    int y;
```

```
} TPoint;
```

```
// Retourne la distance Euclidienne entre 2 points
```

```
void definirX(TPoint* p);
```

```
int donneX(TPoint p);
```

```
float distance(TPoint p, TPoint q);
```

```
...
```

Fichier
point.h

```
/* Implementation du point */
```

```
#include <math.h>
```

```
#include "point.h"
```

```
float distance(TPoint p, TPoint q) {
```

```
    return sqrt((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y));
```

```
}
```

```
...
```

Fichier
point.c

Gestions des exceptions

- Il se peut que le type construit ne soit **pas prévu** pour gérer certains cas
 - On parle de **pré-conditions** ou d'**axiomes**
- On doit pouvoir gérer les cas d'erreurs induits par ces situations
- On énumère des exceptions, des situations où **l'utilisation du type n'est pas légale**
- Du point de vue de l'implémentation, **l'encapsulation facilite la gestion des exceptions**
- Exemples:
 - Valeur d'indice illégale pour un tableau
 - Stockage d'un élément de type incompatible
 - ... *cf* exemples dans ce cours

TAD: Exceptions

Nom du TAD

Rôle, but:

→ Décrire de façon concise le nouveau type de données

Spécification, déclaration:

→ Préciser les contraintes éventuellement nécessaires à son élaboration

Primitives, opérations:

1. **Création, initialisation, construction**
Lister les opérations servant à l'initialisation des instances du type
2. **Modification**
Lister les opérations permettant la modification d'une instance du type (insertion, suppression,...)
3. **Destruction**
Lister les opérations permettant la destruction d'une instance du type
4. **Accès**
Lister les modes d'accès à une instance du type (parcours,...)

Exceptions: [exemple]

- Accéder à une partie inexistante de l'instance du type (4)

- On déclare les cas d'exceptions
- On les associe aux primitives déclenchant cette exception
- On peut documenter la réaction face à ces exceptions

⇒ *cf* exemples dans les structures statiques

TAD: Résumé

- Les TAD sont un outil de **modélisation** indépendant du langage
- Ils permettent une transition vers l'implémentation
- Ils comprennent deux parties essentielles
 - **Interface publique**: les fonctionnalités exposées à l'utilisation
 - Partie **privée**: fonctions internes et non accessibles
- **L'encapsulation** permet de masquer et contrôler l'accès aux attributs du type
- Ils sont associés à des **cas d'exceptions**
- **L'encapsulation** facilite la gestion des exceptions
- Le principe d'interface permet de mettre en place une structure **modulaire**
- L'interface permet de cacher l'implémentation à l'utilisateur.
Celle-ci peut donc varier dans changer l'usage du TAD
- **L'interface peut être complétée** sans modifier les programmes utilisant une version initial de cette interface

A noter: le TAD tel que décrit ici est un prémisses important à la **notion de classe**